SCALABLE BIG DATA SYSTEMS: ARCHITECTURES AND OPTIMIZATIONS

A Thesis
Presented to
The Academic Faculty

by

Kisung Lee

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2015

SCALABLE BIG DATA SYSTEMS: ARCHITECTURES AND OPTIMIZATIONS

Approved by:

Professor Ling Liu, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Karsten Schwan
School of Computer Science
Georgia Institute of Technology

Professor Ed Omiecinski
School of Computer Science
Georgia Institute of Technology

Professor Lakshmish Ramaswamy
Department of Computer Science
University of Georgia

Professor Calton Pu
School of Computer Science
Georgia Institute of Technology

Date Approved: 30 April 2015

To Mom and Dad

ACKNOWLEDGEMENTS

I gratefully acknowledge the invaluable support I have received from my advisor, Professor Ling Liu, over the course of my doctoral study. Her endless enthusiasm for research and unsurpassed energy have inspired me to work on several interesting and practical research problems. In addition to academic advising, she has shared her life lessons with me and always been available whenever I needed help. I have been extremely fortunate to have her "in my corner" as my doctoral advisor, and I hope to pass the lessons I learned from her on to my students in the future.

I would also like to express my thanks to my doctoral dissertation committee members: Professors Ed Omiecinski, Calton Pu, Karsten Schwan, and Lakshmish Ramaswamy. Their insightful comments and suggestions on my research have not only greatly contributed to my thesis but also helped broaden my horizons for my future research. I have also been fortunate to spend three summers at IBM Research T.J. Watson as a summer research intern and experience real-world research and engineering challenges. I express sincere thanks to my IBM mentors and collaborators including Drs. Raghu Ganti, Mudhakar Srivatsa, and Isabelle Rouvellou.

I would like to thank every member of the DiSL Research Group, Databases Laboratory, and Systems Laboratory at Georgia Tech for their collaboration and companionship. It was a great pleasure to work in such a dynamic research environment. I convey special thanks to Qi Zhang, Xianqiang Bao, Yang Zhou, Balaji Palanisamy, Emre Yigitoglu, and Myungcheol Doo for countless research discussions and friendship. I have also been exceptionally fortunate to meet many wonderful friends in Atlanta. I would like to express my special thanks to Jee Eun Park, Joonseok Lee, Soo Kyung Kim, Sangmin Park, and Hannah Kim for the many memories they have given me. I also thank Jane Chisholm for her invaluable help and lessons.

Last but not the least, I do not know how I can thank my mom enough for her unconditional love and countless prayers for me. I am so proud to be her son. I also thank my dad and sister for their love and support. Without the love and prayers of my family, this accomplishment would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Big data analytics has become not just a popular "buzzword" but also a strategic direction in information technology for many enterprises and government organizations. Even though many new computing and storage systems have been developed for big data analytics, scalable big data processing has become more and more challenging as a result of the huge and rapidly growing size of real-world data.

Dedicated to the development of architectures and optimization techniques for scaling big data processing systems, especially in the era of cloud computing, this dissertation makes three unique contributions. First, it introduces a suite of graph partitioning algorithms that can run much faster than existing data distribution methods and inherently scale to the growth of big data. The main idea of these approaches is to partition a big graph by preserving the core computational data structure as much as possible to maximize intra-server computation and minimize inter-server communication. In addition, it proposes a distributed iterative graph computation framework that effectively utilizes secondary storage to maximize access locality and speed up distributed iterative graph computations. The framework not only considerably reduces memory requirements for iterative graph algorithms but also significantly improves the performance of iterative graph computations. Last but not the least, it establishes a suite of optimization techniques for scalable spatial data processing along with three orthogonal dimensions: (i) scalable processing of spatial alarms for mobile users traveling on road networks, (ii) scalable location tagging for improving the quality of Twitter data analytics and prediction accuracy, and (iii) lightweight spatial indexing for enhancing the performance of big spatial data queries.

CHAPTER I

INTRODUCTION

With continued advances in computing and information technology, digital data have grown at an astonishing rate in terms of volume, variety, and velocity. Such big data have huge potential to reveal hidden insights and promote innovation in many business, science, and engineering domains. Even though the application of big data analytics is virtually unlimited, scalable processing of big data becomes more and more challenging because of the huge amount of newly generated data.

There are several key challenges we need to address for scalable processing of big data. First, an important technical challenge faced by many scientists and engineers is how to build efficient big data processing systems and applications that can scale to the rapid growth of digital data in the 21st century. In most cases, conventional data analysis algorithms and computing platforms are inadequate for big data processing because they were primarily designed and developed for running on a single server under centralized computing architecture. In other words, it is not straightforward to run these algorithms and platforms in a distributed computing environment. Second, even though new systems and algorithms are continuously and rapidly being developed for big data analytics, they typically have some limitations on their scalability. One common limitation is in-memory data processing in a distributed computing environment that requires huge main memory to store both input and intermediate data for big data analytics. Unless there is a computing cluster large enough to accommodate such big data, existing systems usually crash in the middle of data processing because of insufficient main memory. Last but not the least, new types of big data, including graph data and spatial data, are being widely used for gaining deep insights into big data. Since most big data systems are designed for structured data, they are usually struggling to handle these new data types in an efficient and scalable manner.

To tackle these challenges of big data processing, this dissertation research is focused on

the development of architectures and optimization techniques for scaling big data processing systems, especially in the era of cloud computing.

1.1 Technical Challenges

We describe the technical challenges to scalable big data processing in more detail as follows.

1.1.1 System Scalability

First, we argue that most existing systems and algorithms for big data processing cannot scale to the rapid growth of real-world data. One representative example is the distributed graph systems for iterative graph algorithms such as PageRank, single-source shortest path, and connected component computations. Existing distributed graph systems, such as Pregel and GraphX, are based on a distributed memory architecture and thus heavily rely on distributed memory for their graph computations. They basically cannot run the iterative graph algorithms unless they have a computing cluster large enough to accommodate both input graph data and intermediate data for graph data analytics. To make matters worse, compared to structured data analytics, it is much more difficult to predict the amount of intermediate data based on the input data size for graph data analytics. In some cases, the size of intermediate data is several orders of magnitude bigger than that of input graph data. Therefore, even though we start with a computing cluster large enough to accommodate the input graph data and potential intermediate data, we may discover that the cluster has insufficient memory to store the intermediate data in the middle of graph data processing and then need to prepare a new cluster with larger memory. This process may be repeated multiple times and thus wasting our time and computing resources (i.e., money) significantly.

Another important limitation in terms of system scalability is that some systems for big data processing are not purely distributed. For example, several graph systems typically called a scalable solution depend on a centralized technique for their graph partitioning. In other words, unless there is a single powerful machine that can run graph partitioning for big graph data, these existing systems cannot even start graph data analytics in a distributed computing environment.

1.1.2 Data Complexity

Another important challenge for scalable big data processing is to support various types of big data, including graph and spatial data, in addition to traditional structured data. These new data types create several new challenges for big data systems. These challenges can be explained using commonly found graph data that are a representative example of new data types. First of all, as we described above, a huge amount of intermediate data is generated during graph analytics, and we cannot easily predict the amount of intermediate data based on the input graph size in most cases. More importantly, graph data have complicated relationships among data entities, and these relationships are essential to gain deep insights into big graph data. However, these complex relationships make it hard to partition the graph for distributed processing. If we use existing distributed data processing frameworks like Apache Hadoop and Spark, the graph will be partitioned without considering these important relationships. Therefore, graph analytics using such distributed systems would be inefficient because we need to find or reconstruct these important relationships in the middle of graph analytics. Last but not the least, real-world graph data have very skewed distribution in terms of the number of connected edges. In other words, there are some vertices with an extremely high degree, and these high-degree vertices make it hard to ensure load balancing during distributed graph processing.

1.1.3 System Complexity

In addition to data complexity, system complexity is also an important challenge for big data systems because systems usually become more complicated when they support more types of data. Since distributed data processing frameworks are typically equipped with several core capabilities including data partitioning, data replication, load balancing, and fault tolerance, algorithms and optimization techniques for supporting new data types should be carefully designed and implemented to minimize any potential overhead to these existing functions. For example, when we want to add support for spatial queries in a distributed storage system, we may consider implementing representative spatial indexing techniques, such as R-trees, in the storage system. However, naive implementation of this approach can

make the storage system more complicated and also increase overhead of existing modules. In addition, even though we handle the system complexity issues carefully for now, most emerging distributed computing systems are being continuously and rapidly updated and thus we should verify that our implementation will still work well for new versions.

1.2 Dissertation Scope and Contributions

To tackle the challenges of big data processing, this dissertation is focused on the development of architectures and a suite of optimization techniques for scaling big data processing systems. This dissertation makes the following contributions to address the technical challenges described above.

1.2.1 Distributed Graph Query Processing

Initially studying the problems and the challenges of distributed processing of big graph data, we develop a distributed RDF (resource description framework) system equipped with semantic hash partitioning. In addition, we develop a general graph partitioning framework for various graph data characteristics and query workloads.

RDF, a standard graph-based model for data exchange on the Web, is being widely used in many scientific projects, governments, and so on. Even though several distributed RDF systems have been proposed, they suffer from high cross-node communication during RDF query processing because they use random partitioning or hash partitioning without taking into account computation correlations among data entities. To tackle this challenge, we develop a distributed RDF system called SHAPE, and propose a scalable partitioning technique for RDF called semantic hash partitioning, which starts with simple hash partitioning and then extends each hash partition through controlled triple replication. Its main goal is to preserve as much of the core computation graph structure as possible to minimize cross-node communication and maximize intra-node computation. We also present an efficient distributed query processing technique by minimizing the cross-node communication based on the semantic hash partitions. The prototype system of SHAPE has been released for public access. In addition, the experimental evaluation on large graphs with hundreds of millions of vertices and billions of edges has shown that SHAPE, which can scale to large

graphs with varying sizes and complexity, is more efficient than existing distributed RDF systems.

Even though graph partitioning is essential for distributed graph processing in the cloud, effectively partitioning a big graph to efficiently process graph queries is challenging because of high data correlations, high heterogeneity, and the highly skewed distribution of graph data. Furthermore, our experiments reveal that existing graph partitioners, mostly based on minimum cut, cannot scale to large graphs with more than a half billion edges. To tackle this challenge, we develop a distributed graph partitioning framework called VB-Partitioner, which supports efficient graph query processing for various graph data characteristics and query workloads. Our framework consists of three main phases. First, it generates partitioning building blocks based on structural correlations among vertices. Second, it groups the building blocks to construct a set of partitions. Since one grouping technique cannot fit all, we propose three different grouping techniques for this phase. Finally, while running graph queries based on the generated partitions, it supports two types of distributed graph query processing: (1) intra-partition processing, in which compute nodes do not communicate, and (2) inter-partition processing, in which coordination among compute nodes is required to join the intermediate results. The experimental results show that VB-Partitioner, which can scale to large graphs, significantly outperforms the popular random block-based graph partitioning in terms of query latency.

1.2.2 Distributed Iterative Graph Computations

Iterative graph computations, such as PageRank, connected component, and single-source shortest path computations, have been widely used to analyze large graphs and thus to derive profound insights from a huge number of explicit and implicit relationships among entities. Existing distributed graph systems for iterative graph computations heavily rely on distributed memory and thus suffer from poor scalability when the compute cluster can no longer hold the graph and all the intermediate results in memory. To address this challenge, we develop GraphMap, a distributed iterative graph computation framework that effectively utilizes secondary storage to maximize access locality and speed up distributed

iterative graph computations. We distinguish read-only graph data from mutable graph data and store the read-only data on disk. To maximize sequential disk access and minimize random disk access, we propose locality-optimized data placement based on a two-level graph partitioning algorithm. Furthermore, we develop locality-based optimization, which dynamically chooses between sequential disk access and random disk access based on the computation loads of each iteration for each worker machine. Compared to Apache Hama, the first prototype of GraphMap not only considerably reduces the memory requirement for iterative graph algorithms but also significantly improves the iterative graph computation performance.

1.2.3 Spatial Data Processing

We also explore the challenges of big spatial data processing by proposing efficient spatial data management techniques along with three orthogonal directions.

First, we develop RoadAlarm, a scalable system for managing and supporting spatial alarms for mobile users traveling on road networks. Spatial alarms are location-based reminders that can inform mobile users when they arrive at a user-specified spatial location of interest such as a grocery store. A key technical challenge for a large-scale spatial alarm system is the ability to provide high performance and high accuracy for spatial alarm evaluation. This system offers a suite of alarm processing and optimization techniques that minimize the amount of wakeups at mobile clients to save energy while reducing the amount of unnecessary alarm checks at the server to improve the performance of servers and the accuracy of alarm evaluations. We evaluate RoadAlarm techniques over large mobile traces and compare them with existing techniques by varying the number of mobile clients, the number of alarms, and the size of the road networks. The RoadAlarm approach outperforms existing approaches by orders of magnitude in terms of both server performance and client energy consumption. We also build a simulation-based demo to show the working of the RoadAlarm system using GTMobiSIM.

Second, we develop a Twitter location prediction framework by utilizing another social network specialized in locations. Even though the location information of a social network is

invaluable, many social network users have been reluctant to adopt the geo-tagging feature of social networks. To solve this location sparseness problem, we propose the framework to predict the location of each tweet. This problem is much more challenging than predicting the location of each user because each tweet has very short textual data (up to 140 characters). Furthermore, the goal of this framework is to predict the fine-grained location (e.g., latitude and longitude) of each tweet, instead of the city- or zip code-level location. We build probabilistic models for locations using data from Foursquare, which is another social network specialized in locations, instead of noisy data from Twitter. To increase the accuracy of prediction, we evaluate various ranking methods, smoothing techniques, and language models. In addition, using machine-learning techniques, we develop classification models that filter out unpredictable tweets.

Third, we develop a lightweight spatial indexing technique for big spatial data by utilizing a hierarchical data structure. Even though several techniques that support spatial queries for distributed storage systems such as HDFS and HBase have been proposed, most require internal modification of existing systems and thus increase the complexity and overhead of the systems. We propose an efficient and lightweight spatial index for big data stored in distributed storage systems. The index, based on a hierarchical spatial data structure, has several advantages. First, it can be easily applied to existing storage systems without modifying their internal implementation. Second, it provides simple yet highly efficient filtering for spatial objects because it uses prefix matching to find relevant spatial objects. Third, it supports the customizable control of the index size for various applications. Our experimental evaluation shows that our approach can significantly improve the search performance of big spatial data and easily be applied to existing storage systems without modifying their internal implementation. A prototype implementation on top of HBase and an RDF store is developed to show the effectiveness and efficiency of our index.

1.3 Dissertation Organization

This dissertation consists of several chapters and each chapter addresses one or more of the problems described above. In each chapter, we introduce the background of the problem

being addressed, describe related work, and present our solution techniques followed by experimental evaluation. This dissertation is organized as follows.

In Chapter 2, we present a new distributed RDF system, called SHAPE, to improve the performance of distributed RDF query processing. Equipped with a scalable partitioning technique and an efficient distributed query processing technique, SHAPE, which can scale to large graphs with varying sizes and complexity, is more efficient than existing distributed RDF systems.

In Chapter 3, we proposes a distributed graph partitioning framework called VB-PARTITIONER, which supports efficient graph query processing for various graph data characteristics and query workloads. Equipped with three different grouping techniques, VB-PARTITIONER significantly outperforms the popular random block-based graph partitioning in terms of query latency.

In Chapter 4, we introduce a distributed iterative graph computation framework called GRAPHMAP, which effectively utilizes secondary storage to maximize access locality and speed up distributed iterative graph computations. The framework not only considerably reduces memory requirements for iterative graph algorithms but also significantly improves the performance of iterative graph computations.

In Chapter 5, we develop a road network-aware spatial alarm processing system called ROADALARM. ROADALARM offers a suite of alarm processing and optimization techniques that minimize the amount of wakeups at mobile clients to save energy while reducing the amount of unnecessary alarm checks at the server to improve the server performance and the accuracy of alarm evaluations.

In Chapter 6, we present a framework to predict the location of each tweet on Twitter. We build probabilistic models for locations using data from Foursquare, which is another social network specialized in locations, instead of noisy data from Twitter.

In Chapter 7, we introduce an efficient and lightweight spatial index for big data stored in distributed storage systems. Based on a hierarchical spatial data structure, the index can be easily applied to existing storage systems without modifying their internal implementation.

In Chapter 8, we summarize the main contributions of this dissertation and discuss our

future research directions.

CHAPTER II

SHAPE: DISTRIBUTED RDF SYSTEM WITH SEMANTIC HASH PARTITIONING

Massive volumes of big RDF data are growing beyond the performance capacity of conventional RDF data management systems operating on a single node. Applications using large RDF data demand efficient data partitioning solutions for supporting RDF data access on a cluster of compute nodes. In this chapter we present a novel semantic hash partitioning approach and implement a Semantic HAsh Partitioning-Enabled distributed RDF data management system, called SHAPE. This chapter makes three original contributions. First, the semantic hash partitioning approach we propose extends the simple hash partitioning method through direction-based triple groups and direction-based triple replications. The latter enhances the former by controlled data replication through intelligent utilization of data access locality, such that queries over big RDF graphs can be processed with zero or very small amount of inter-machine communication cost. Second, we generate locality-optimized query execution plans that are more efficient than popular multi-node RDF data management systems by effectively minimizing the inter-machine communication cost for query processing. Third but not the least, we provide a suite of locality-aware optimization techniques to further reduce the partition size and cut down on the inter-machine communication cost during distributed query processing. Experimental results show that our system scales well and can process big RDF datasets more efficiently than existing approaches.

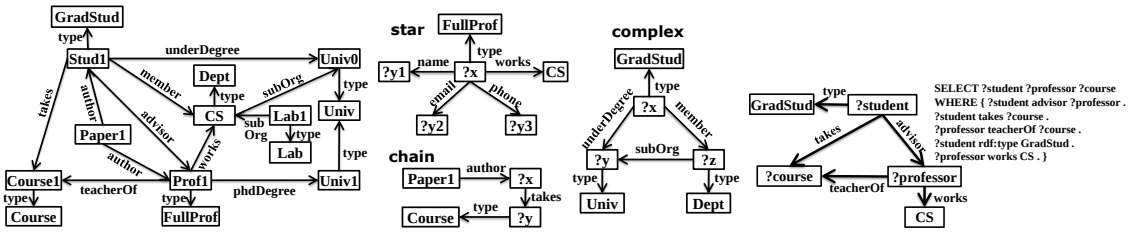
2.1 Introduction

The creation of RDF (resource description framework) [17] data is escalating at an unprecedented rate, led by the semantic web community and Linked Open Data initiatives [14]. On one hand, the continuous explosion of RDF data opens door for new innovations in big data and Semantic Web initiatives, but on the other hand, it easily overwhelms the memory and computation resources on commodity servers and causes performance bottlenecks in

many existing RDF stores with query interfaces such as SPARQL [19]. Furthermore, many scientific and commercial online services must answer queries over big RDF data in near real time, and achieving fast query response time requires careful partitioning and distribution of big RDF data across a cluster of servers.

A number of distributed RDF systems are using Hadoop MapReduce as their query execution layer to coordinate query processing across a cluster of server nodes. Several independent studies have shown that a sharp difference in query performance is observed between queries that are processed completely in parallel without any coordination among server nodes and queries that require even a small amount of coordination. When the size of intermediate results is large, the inter-node communication cost for transferring intermediate results of queries across multiple server nodes can be prohibitively high. Therefore, we argue that a scalable RDF data partitioning approach should be able to partition big RDF data into performance-optimized partitions such that the number of queries that hit partition boundaries is minimized and the cost of multiple rounds of data shipping across a cluster of sever nodes is eliminated or reduced significantly.

In this chapter we present a semantic hash partitioning approach that combines locality-optimized RDF graph partitioning with cost-aware query partitioning for scaling queries over big RDF graphs. At the data partitioning phase, we develop a semantic hash partitioning method that utilizes access locality to partition big RDF graphs across multiple compute nodes by maximizing the intra-partition processing capability and minimizing the inter-partition communication cost. Our semantic hash partitioning approach introduces direction-based triple groups and direction-based triple replications to enhance the baseline hash partitioning algorithm by controlled data replication through intelligent utilization of data access locality. We also provide a suite of semantic optimization techniques to further reduce the partition size and increase the opportunities for intra-partition processing. As a result, queries over big RDF graphs can be processed with zero or very small amount of inter-partition communication cost. At the cost-aware query partitioning phase, we generate locality-optimized query execution plans that can effectively minimize the inter-partition



(a) RDF Graph (b) SPARQL Query Graphs (c) Example SPARQL query

Figure 1: RDF and SPARQL

communication cost for distributed query processing and are more efficient than those produced by popular multi-node RDF data management systems. To validate our semantic hash partitioning architecture, we develop SHAPE, a Semantic HAsh Partitioning-Enabled distributed RDF data management system. We experimentally evaluate our system to understand the effects of various system parameters and compare against other popular RDF data partitioning schemes, such as simple hash partitioning and min-cut graph partitioning. Experimental results show that our system scales well and can process big RDF datasets more efficiently than existing approaches. Although this chapter focuses on RDF data and SPARQL queries, we conjecture that many of our technical developments are applicable to scaling queries and subgraph matching over general applications of big graphs.

The rest of the chapter proceeds as follows. We give a brief overview of RDF, SPARQL, and the related work in Section 2.2. Section 2.3 describes the SHAPE system architecture that implements the semantic hash partitioning. We present the locality-optimized semantic hash partitioning scheme in Section 2.4 and the partition-aware distributed query processing mechanisms in Section 2.5. We report our experimental results in Section 2.6 and conclude the chapter in Section 2.7.

2.2 Preliminary

2.2.1 RDF and SPARQL

RDF is a standard data model proposed by World Wide Web Consortium (W3C). An RDF dataset consists of RDF triples, and each triple has a subject, a predicate and an object, representing a relationship, denoted by the predicate, between the subject and the object. An RDF dataset forms a directed, labeled RDF graph, where subjects and objects are

vertices and predicates are labels on the directed edges, each emanating from its subject vertex to its object vertex. The schema-free model makes RDF attractive as a flexible mechanism for describing entities and relationships among entities. Fig. 1(a) shows an example RDF graph, extracted from the Lehigh University Benchmark (LUBM) [47].

SPARQL [19] is a SQL-like standard query language for RDF, recommended by W3C. SPARQL queries consist of triple patterns, in which subject, predicate and object may be a variable. A SPARQL query is said to match subgraphs of the RDF data when the terms in the subgraphs may be substituted for the variables. Processing a SPARQL query Q involves graph pattern matching, and the result of Q is a set of subgraphs in the big RDF graph, which match the triple patterns of Q.

SPARQL queries can be categorized into star, chain and complex queries as shown in Fig. 1(b). *Star* queries often consist of subject-subject joins, and each join variable is the subject of all the triple patterns involved. *Chain* queries often consist of subject-object joins (i.e., the subject of a triple pattern is joined to the object of another triple pattern), and their triple patterns are connected one by one like a chain. We refer to the remaining queries, which are combinations of star and chain queries, as *complex* queries.

2.2.2 Related Work

Data partitioning is an important problem with applications in many areas. Hash partitioning is one of the dominating approaches in RDF graph partitioning. It divides an RDF graph into smaller and similar sized partitions by hashing over the subject, predicate or object of RDF triples. We classify existing distributed RDF systems into two categories based on how the RDF dataset is partitioned and how partitions are stored and accessed.

The first category generally partitions an RDF dataset across multiple servers using horizontal (random) partitioning, stores partitions using distributed file systems such as Hadoop Distributed File System (HDFS), and processes queries by parallel access to the clustered servers using distributed programming model such as Hadoop MapReduce [97, 56]. SHARD [97] directly stores RDF triples in HDFS as flat text files and runs one Hadoop job for each clause (triple pattern) of a SPARQL query. [56] stores RDF triples in HDFS by

hashing on predicates and runs one Hadoop job for each join of a SPARQL query. Existing approaches in this category suffer from prohibitively high inter-node communication cost for processing queries.

The second category partitions an RDF dataset across multiple nodes using hash partitioning on subject, object, predicate or any combination of them. However, the partitions are stored locally in a database, such as a key-value store like HBase or an RDF store like RDF-3X [86] and accessed via a local query interface. In contrast to the first type of systems, these systems only resort to distributed computing frameworks, such as Hadoop MapReduce, to perform cross-server coordination and data transfer required for distributed query execution, such as joins of intermediate query results from two or more partition servers [42, 50, 88, 68]. Concretely, Virtuoso Cluster [42], YARS2 [50], Clustered TDB [88] and CumulusRDF [68] are distributed RDF systems that use simple hashing as their triple partitioning strategy, but they differ from one another in terms of their index structures. Virtuoso Cluster partitions each index of all RDBMS tables containing RDF data using hashing. YARS2 uses hashing on the first element of all six alternately ordered indices to distribute triples to all servers. Clustered TDB uses hashing on subject, object and predicate to distribute each triple three times to the cluster of servers. CumulusRDF distributes three alternately ordered indices using a key-value store. Surprisingly, none of the existing data partitioning techniques by design aim at minimizing the amount of inter-partition coordination and data transfer involved in distributed query processing. Thus, most existing work suffers from the high cost of cross-server coordination and data transfer for complex queries. Such heavy inter-partition communication incurs excessive network I/O operations, leading to long query latencies.

Graph partitioning has been studied extensively in several communities for decades [52, 60]. A typical graph partitioner divides a graph into smaller partitions that have minimum connections between them, as adopted by METIS [60, 15] or Chaco [52]. Various efforts on graph partitioning have been dedicated to partitioning a graph into similar sized partitions such that the workload of servers holding these partitions will be better balanced. [55] promotes the use of min-cut based graph partitioner for distributing big RDF

data across a cluster of machines. It shows experimentally that min-cut based graph partitioning outperforms the simple hash partitioning approach. However, the main weaknesses of existing graph partitioners are the high overhead of loading the RDF data into the data format of graph partitioners and the poor scalability to large datasets. For example, we show in Section 2.6 that it is time consuming to load large RDF datasets to a graph partitioner and the partitioner also crashes when RDF datasets exceed a half billion triples. Orthogonal to graph partitioning efforts such as min-cut algorithms, several vertex-centric programming models are proposed for efficient graph processing on a cluster of commodity servers [80, 70] or for minimizing disk IOs required by in-memory graph computation [67]. Concretely, [80, 67] are known for their iterative graph computation techniques that can speed up certain types of graph computations. The techniques developed in [70] partition heterogeneous graphs by constructing customizable types of vertex blocks.

In comparison, this is the first work, to the best of our knowledge, which introduces a semantic hash partitioning method combined with a locality-aware query partitioning method. The semantic hash partitioning method extends simple hash partitioning by combining direction-based triple grouping with direction-based triple replication. The locality-aware query partitioning method generates semantic hash partition-aware query plans, which minimize inter-partition communication cost for distributed query processing.

2.3 Overview

We implement the first prototype system of our semantic hash partitioning method on top of Hadoop MapReduce with the master server as the coordinator and the set of slave servers as the workers. Fig. 2 shows a sketch of our system architecture.

Data partitioning. RDF triples are fetched into the data partitioning module installed on the master server, which partitions the data stored across the set of slave servers. To work with big data that exceeds the performance capacity (e.g., memory, CPU) of a single server, we provide a distributed implementation of our semantic hash partitioning algorithm to perform data partitioning using a cluster of servers. The semantic hash partitioner performs three main tasks: (i) *Pre-partition optimizer* prepares the input RDF dataset for hash

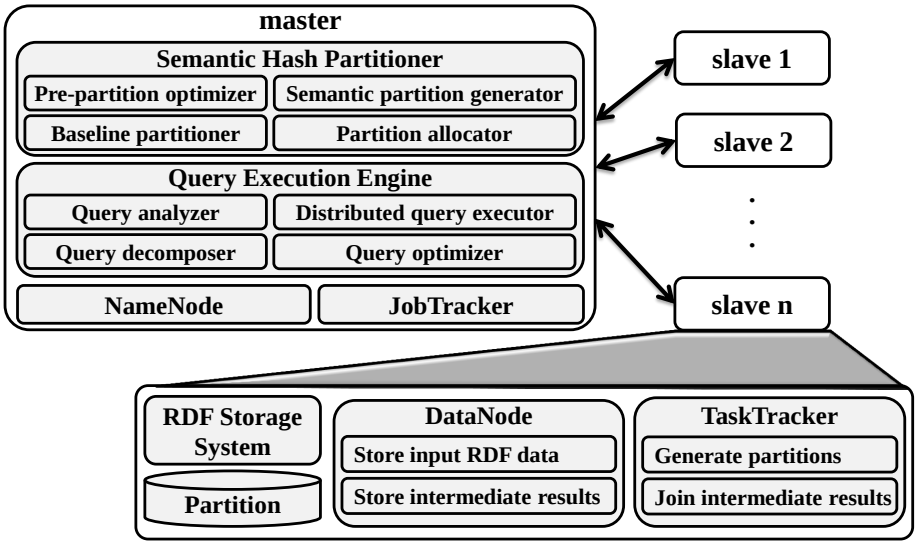


Figure 2: SHAPE System Architecture

partitioning, aiming at increasing the access locality of each baseline partition generated in the next step. (ii) *Baseline hash partition generator* uses a simple hash partitioner to create a set of baseline hash partitions. In the first prototype implementation, we set the number of partitions to be exactly the number of available slave servers. (iii) *Semantic hash partition generator* utilizes the triple replication policies (see Section 2.4) to determine how to expand each baseline partition to generate its semantic hash partition with high access locality. We utilize the selective triple replication optimization technique to balance between the access locality and the partition size. On each slave server, either an RDF-specific storage system or a relational DBMS can be installed to store the partition generated by the data partitioning algorithms. It also processes SPARQL queries over the local partition stored in the slave server and generates partial (or intermediate) results. RDF-3X [86] is installed on each slave server of the cluster in SHAPE.

Distributed query processing. The master server also serves as the interface for SPARQL queries and performs distributed query execution planning for each query received. We categorize SPARQL query processing on a cluster of servers into two types: *intra*-partition processing and *inter*-partition processing.

By intra-partition processing, we mean that a query Q can be fully executed in parallel on each server by locally searching the subgraphs matching the triple patterns of Q, without

any inter-partition coordination. The only inter-server communication cost required to process Q is for the master server to send Q to each slave server and for each slave server to send its local matching results to the master server, which simply merges the partial results received from all slave servers to generate the final results of Q.

By inter-partition processing, we mean that a query Q as a whole cannot be executed on any partition server, and it needs to be decomposed into a set of subqueries such that each subquery can be evaluated by intra-partition processing. Thus, the processing of Q requires multiple rounds of coordination and data transfer across a set of partition servers. In contrast to intra-partition processing, the communication cost for inter-partition processing can be extremely high, especially when the number of subqueries is not small and the size of intermediate results to be transferred across a network of partition servers is large.

2.4 Semantic Hash Partitioning

The semantic hash partitioning algorithm performs data partitioning in three main steps. First, we build a set of triple groups that are baseline building blocks for semantic hash partitioning. Second, we group the baseline building blocks to generate baseline hash partitions. To further increase the access locality of the baseline building blocks, we also develop an RDF-specific optimization technique that applies URI hierarchy-based grouping to merge those triple groups whose anchor vertices share the same URI prefix prior to generating the baseline hash partitions. Third, we generate k-hop semantic hash partitions that expand each baseline hash partition via controlled triple replication. To further balance the amount of triple replication and the efficiency of query processing, we also develop the `rdf:type`-based triple filter during the k-hop triple replication. To ease the readability, we first describe the three core tasks and then discuss the two optimizations at the end of this section.

2.4.1 Building Triple Groups

An intuitive way to partition a large RDF dataset is to group a set of triples anchored at the same subject or object vertex and place the grouped triples in the same partition. We call such groups *triple groups*, each with an anchor vertex. Triple groups are used as

baseline building blocks for our semantic hash partitioning. An obvious advantage of using the triple groups as baseline building blocks is that star queries can be efficiently executed in parallel using solely intra-partition processing at each sever because it is guaranteed that all required triples, from each anchor vertex, to evaluate a star query are located in the same partition.

Definition 1 (RDF Graph) An RDF graph is a directed, labeled multigraph, denoted as $G = (V, E, \Sigma_E, l_E)$ where V is a set of vertices and E is a multiset of directed edges (i.e., ordered pairs of vertices). $(u, v) \in E$ denotes a directed edge from u to v. Σ_E is a set of available labels (i.e., predicates) for edges and l_E is a map from an edge to its label $(E \rightarrow \Sigma_E)$.

In RDF datasets, multiple triples may have the same subject and object and thus E is a multiset instead of a set. Also the size of E ($|E|$) represents the total number of triples in the RDF graph G.

For each vertex v in a given RDF graph, we define three types of triple groups based on the role of v with respect to the triples anchored at v: (i) *subject-based* triple group (*s-TG*) of v consists of those triples in which their subject is v (i.e., outgoing edges from v) (ii) *object-based* triple group (*o-TG*) of v consists of those triples in which their object is v (i.e., incoming edges to v) (iii) *subject-object-based* triple group (*so-TG*) of v consists of those triples in which their subject or object is v (i.e., all connected edges of v). We formally define triple groups as follows.

Definition 2 (Triple Group) Given an RDF graph $G = (V, E, \Sigma_E, l_E)$, *s-TG* of vertex $v \in V$ is a set of triples in which their subject is v, denoted by *s-TG(v)* $= \{(u, w)|(u, w) \in E, u = v\}$. We call v the *anchor* vertex of *s-TG(v)*. Similarly, *o-TG* and *so-TG* of v are defined as *o-TG(v)* $= \{(u, w)|(u, w) \in E, w = v\}$ and *so-TG(v)* $= \{(u, w)|(u, w) \in E, v \in \{u, w\}\}$ respectively.

We generate a triple group for each vertex in an RDF graph and use the set of generated triple groups as baseline building blocks to generate k-hop semantic hash partitions. The

subject-based triple groups are anchored at subject of the triples and are efficient for *subject-based* star queries in which the center vertex is the subject of all triple patterns (i.e., subject-subject joins). The total number of *s-TG* equals to the total number of distinct subjects. Similarly, *o-TG* and *so-TG* are efficient for *object-based* star queries (i.e., object-object joins), in which the center vertex is the object of all triple patterns, and *subject-object-based* star queries, in which the center vertex is the subject of some triple patterns and object of the other triple patterns (i.e., there exists at least one subject-object join) respectively.

2.4.2 Constructing Baseline Hash Partitions

The *baseline* hash partitioning takes the triple groups generated in the first step, applies a hash function on the anchor vertex of each triple group, and places those triple groups having the same hash value in the same partition. We can view the baseline partitioning as a technique to bundle different triple groups into one partition. With three types of triple groups, we can construct three types of baseline partitions: subject-based partitions, object-based partitions, and subject-object-based partitions.

Definition 3 (Baseline hash partitions) Let $G = (V, E, \Sigma_E, l_E)$ denote an RDF graph and $TG(v)$ denote the triple group anchored at vertex $v \in V$. The baseline hash partitioning P of graph G results in a set of n partitions, denoted by $\{P_1, P_2, \ldots, P_n\}$ such that $P_i = (V_i, E_i, \Sigma_{E_i}, l_{E_i})$, $\bigcup_i V_i = V$, $\bigcup_i E_i = E$. If $V_i = \{v|hash(v) = i, v \in V\} \bigcup \{w|(v,w) \in TG(v), hash(v) = i, v \in V\}$ and $E_i = \{(v,w)|v,w \in V_i, (v,w) \in E\}$, we call the baseline partitioning P the *s-TG* hash partitioning. If $V_i = \{v|hash(v) = i, v \in V\} \bigcup \{u|(u,v) \in TG(v), hash(v) = i, v \in V\}$ and $E_i = \{(u,v)|u,v \in V_i, (u,v) \in E\}$, we call the baseline partitioning P the *o-TG* hash partitioning. In the above two cases, $E_i \bigcap E_j = \emptyset$ for $1 \leq i, j \leq n$, $i \neq j$. If $V_i = \{v|hash(v) = i, v \in V\} \bigcup \{w|(v,w) \in TG(v) \vee (w,v) \in TG(v), hash(v) = i, v \in V\}$ and $E_i = \{(v,w)|v,w \in V_i, (v,w) \in E\}$, we call the baseline partitioning P the *so-TG* hash partitioning.

We can verify the correctness of the baseline partitioning by checking the full coverage of baseline partitions and the disjoint properties for subject-based and object-based baseline partitions. In addition, we can further improve the partition balance across a cluster of

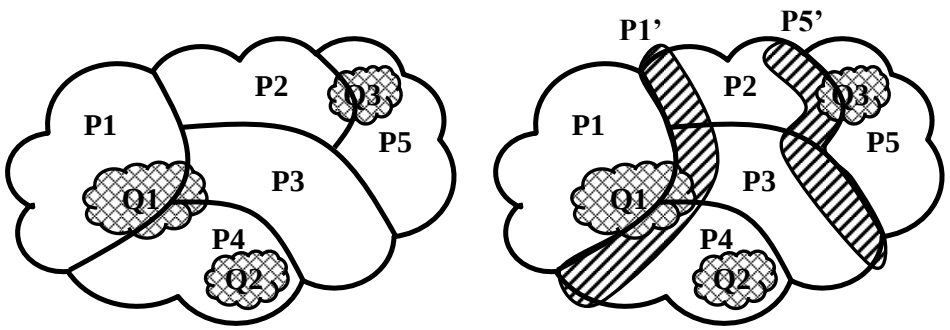
servers by fine-tuning of triple groups with high degree anchor vertices. We omit further discussion on the correctness verification and quality assurance step in this chapter.

2.4.3 Generating Semantic Hash Partitions

Using a hash function to map triple groups to baseline partitions has two advantages. It is simple, and it generates well balanced partitions. However, a serious weakness of simple hash-based partitioning is the poor performance for complex non-star queries.

Considering a complex SPARQL query asking the list of graduate students who have taken a course taught by their CS advisor in Fig. 1(c), its query graph consists of two star query patterns chained together: one consists of three triple patterns emanating from variable vertex `?student`, and the other consists of two triple patterns emanating from variable vertex `?professor`. Assuming that the original RDF data in Fig. 1(a) is partitioned using the simple hash partitioning based on *s-TG*s, we know that the triples with predicates `advisor` and `takes` emanating from their subject vertex `Stud1` are located in the same partition. However, it is highly likely that the triple `teacherOf` and the triple `works` emanating from a different but related subject vertex `Prof1`, the advisor of the student `Stud1`, are located in a different partition, because the hash value for `Stud1` is different from the hash value of `Prof1`. Thus, this complex query needs to be evaluated by performing inter-partition processing, which involves splitting the query into a set of subqueries as well as cross-server communication and data shipping. Assume that we choose to decompose the query into the following two subqueries: 1) SELECT ?student ?professor ?course WHERE {?student advisor ?professor . ?student takes ?course . ?student rdf:type GradStud . } 2) SELECT ?professor ?course WHERE {?professor teacherOf ?course . ?professor works CS .}. Although each subquery can be performed in parallel on all partition servers, we need to ship the intermediate results generated from each subquery across a network of partition servers in order to join the intermediate results of the subqueries, which can lead to a high cost of inter-server communication.

Taking a closer look at the query graph in Fig.1(c), it is intuitive to observe that if the triples emanating from the vertex `Stud1` and the triples emanating from its one hop neighbor



(a) Baseline partitions (b) Expanded Partitions

Figure 3: Partition Expansion

vertex `Prof1` are residing in the same partition, we can effectively eliminate the inter-partition processing cost and evaluate this complex query by only intra-partition processing. This motivates us to develop a locality-aware semantic hash partitioning algorithm through hop-based controlled triple replication.

2.4.3.1 Hop-based Triple Replication

The main goal of using hop-based triple replication is to create a set of semantic hash partitions such that the number of queries that can be evaluated by intra-partition processing is increased. In contrast, with the baseline partitions only star queries can be guaranteed for intra-partition processing.

Fig. 3 presents an intuitive illustration of the concept and benefit of the semantic hash partitioning. By the baseline hash partitioning, we have five baseline partitions P1, P2, P3, P4, and P5 and three queries shown in Fig. 3(a). For brevity, we assume that the baseline partitions are generated using *s-TG*s or *o-TG*s in which each triple is included in only one triple group. Clearly, Q_2 is an intra-partition query and Q_1 and Q_3 are inter-partition queries. Evaluating Q_1 requires to access triples located in and nearby the boundaries of the three partitions: P1, P3, and P4. One way to process Q_1 is to use the baseline partitions. Thus, Q_1 should be split into three subqueries, and upon completion of the subqueries, their intermediate results are joined using Hadoop jobs. The communication cost for inter-partition processing depends on the number of subqueries, the size of the intermediate results, and the size of the cluster (i.e., number of partition servers involved).

Alternatively, we can expand the triple groups in each baseline partition by using hop-based triple replication and execute queries over the semantic hash partitions instead. In Fig. 3(b), the shaded regions, P1' and P5', represent a set of replicated triples added to partition P1 and P5 respectively. Thus, P1 is replaced by its semantic hash partition, denoted by P1 $\bigcup$ P1'. Similarly, P5 is replaced by P5 $\bigcup$ P5'. With the semantic hash partitions, all three queries can be executed by intra-partition processing without any coordination with other partitions and any join of intermediate results, because all triples required to evaluate the queries are located in the expanded partition.

Before we formally introduce the k-hop semantic hash partitioning, we first define some basic concepts of RDF graphs.

Definition 4 (Path) Given an RDF graph $G = (V, E, \Sigma_E, l_E)$, a **path** from vertex $u \in V$ to another vertex $w \in V$ is a sequence of vertices, denoted by $v_0, v_1, \ldots, v_k$, such that $v_0 = u$, $v_k = w$, $\forall m \in [0, k-1] : (v_m, v_{m+1}) \in E$. We also call this path the *forward direction path*. A *reverse direction path* from vertex u to vertex w is a sequence of vertices, denoted by $v_0, v_1, \ldots, v_k$, such that $v_0 = u$, $v_k = w$, $\forall m \in [0, k-1] : (v_{m+1}, v_m) \in E$. A *bidirection path* from vertex u to vertex w is a sequence of vertices, denoted by $v_0, v_1, \ldots, v_k$, such that $v_0 = u$, $v_k = w$, $\forall m \in [0, k-1] : (v_m, v_{m+1}) \in E$ or $(v_{m+1}, v_m) \in E$. The **length** of the path $v_0, v_1, \ldots, v_k$ is k.

Definition 5 (Hop count) Given an RDF graph $G = (V, E, \Sigma_E, l_E)$, we define the **hop count** from vertex $u \in V$ to vertex $v \in V$, denoted by $hop(u, v)$, as the minimum length of all possible forward direction paths from u to v. We also define the hop count from vertex u to edge $(v, w) \in E$, denoted by $hop(u, vw)$, as "$1 + hop(u, v)$". The reverse hop count from vertex u to vertex v, $reverse\_hop(u, v)$, is the minimum length of all possible reverse direction paths from u to v. The bidirection hop count from vertex u to vertex v, $bidirection\_hop(u, v)$, is the minimum length of all possible bidirection paths between u to v. The hop count $hop(u, v)$ is zero if $u = v$ and ∞ if there is no forward direction path from u to v. Similar exceptions exist for $reverse\_hop(u, v)$ and $bidirection\_hop(u, v)$.

Now we introduce k-hop expansion to control the level of triple replication and balance

between the query performance and the cost of storage. Concretely, each *expanded* partition will contain all triples that are within k hops from *any* anchor vertex of its triple groups. k is a system-defined parameter and $k = 2$ is the default setting in our first prototype. One way to optimize the setting of k is to utilize the statistics collected from representative historical queries such as frequent query patterns.

We support three approaches to generate k-hop semantic hash partitions based on the direction of triple expansion: i) forward direction-based, ii) reverse direction-based, and iii) bidirection-based. The main advantage of using direction-based triple replication is to enable us to selectively replicate the triples within k hops. This selective replication strategy offers a configurable and customizable means for users and applications of our semantic hash partitioner to control the amount of triple replications desired. This is especially useful when considering a better tradeoff between the gain of minimizing inter-partition processing and the cost of local storage and local query processing. Furthermore, by enabling direction-based triple expansion, we provide k-hop semantic hash partitioning with a flexible combination of tripe groups of different types and k-hop triple expansion to baseline partitions along different directions.

Let $G = (V, E, \Sigma_E, l_E)$ be the RDF graph of the original dataset and $\{P_1, P_2, \ldots, P_m\}$ denote the baseline partitions on G. We formally define the k-hop *forward* semantic hash partitions as follows.

Definition 6 (k-hop *forward* semantic hash partition) The k-hop forward semantic hash partitions on G are expanded partitions from $\{P_1, P_2, \ldots, P_m\}$, by adding (replicating) triples that are within k hops from any *anchor* vertex in each baseline partition along the *forward* direction, denoted by $\{P_1^k, P_2^k, \ldots, P_m^k\}$, where each baseline partition $P_i = (V_i, E_i, \Sigma_{E_i}, l_{E_i})$ is expanded into $P_i^k = (V_i^k, E_i^k, \Sigma_{E_i^k}, l_{E_i^k})$ such that $E_i^k = \{e | e \in E, \exists v_{anchor} \in V_i : hash(v_{anchor}) = i$ and $hop(v_{anchor}, e) \leq k\}$, and $V_i^k = \{v | (v, v') \in E_i^k$ or $(v'', v) \in E_i^k\}$.

We omit the formal definitions of the k-hop *reverse* and *bidirection* semantic hash partitions, in which the only difference is using $reverse\_hop(v_{anchor}, e)$ and $bidirection\_hop(v_{anchor}, e)$, instead of using $hop(v_{anchor}, e)$, respectively.

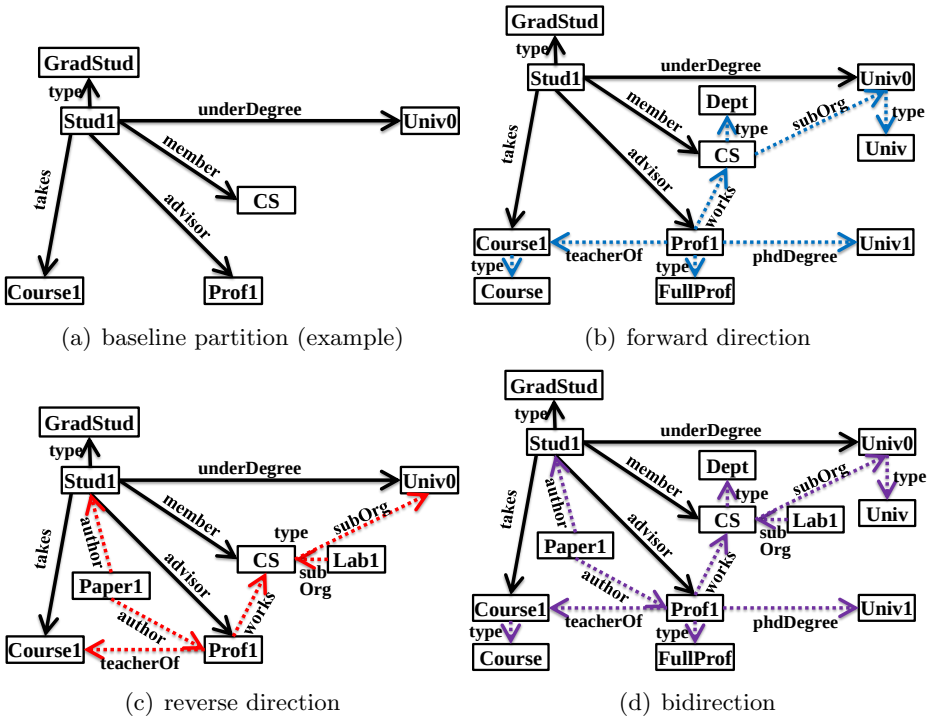


Figure 4: Semantic Hash Partitions from `Stud1`

Fig. 4 illustrates three direction-based 2-hop expansions from a triple group with anchor vertex `Stud1` shown in Fig. 4(a). Fig. 4(b) shows the 2-hop forward semantic hash partition, where dotted edges represent replicated triples by 2-hop expansion from the baseline partition. Fig. 4(c) shows the 2-hop reverse semantic hash partition (i.e., from object to subject). Fig. 4(d) shows the semantic hash partition generated by 2-hop bidirection expansion from `Stud1`.

2.4.3.2 Benefits of k-hop semantic hash partitions

The main idea of the semantic hash partitioning approach is to use a flexible triple replication scheme to maximize intra-partition processing and minimize inter-partition processing for RDF queries. Compared to existing data partitioning algorithms that produce disjoint partitions, the biggest advantage of using the k-hop semantic hash partitioning is that, by selectively replicating some triples across multiple partitions, more queries can be executed using intra-partition processing.

We employ the concept of eccentricity, radius and center vertex to formally characterize

the benefits of the k-hop semantic hash partitioning scheme. Let $G = (V, E, \Sigma_E, l_E)$ denote an RDF graph.

Definition 7 (Eccentricity) The **eccentricity** ϵ of a vertex $v \in V$ is the greatest bidirection hop count from v to any edge in G and formally defined as follows:

$$\epsilon(v) = \max_{e \in E} bidirection\_hop(v, e)$$

The eccentricity of a vertex in an RDF graph shows how far a vertex is from the vertex most distant from it in the graph. In the above definition, if we use the forward or reverse hop count instead, we can obtain the *forward* or *reverse* eccentricity respectively.

Definition 8 (Radius and Center vertex) We define the **radius** of G, $r(G)$, as the minimum (bidirection) eccentricity of any vertex $v \in V$. The **center vertices** of G are the vertices whose (bidirection) eccentricity is equal to the radius of G.

$$r(G) = \min_{v \in V} \epsilon(v),\ center(G) = \{v | v \in V, \epsilon(v) = r(G)\}$$

When the *forward* or *reverse* eccentricity is used to define the radius of an RDF graph G, we refer to this radius as the forward or reverse direction radius respectively.

Now we use the query radius to formalize the gain of the semantic hash partitioning. Given a query Q issued over a set of k-hop semantic hash partitions, if the *radius* of Q's query graph is equal to or less than k, then Q can be executed on the partitions by using intra-partition processing.

Theorem 1 Let $\{P_1^k, P_2^k, \ldots, P_m^k\}$ denote the semantic hash partitions of G, generated by k-hop expansion from the baseline partitions $\{P_1, P_2, \ldots, P_m\}$ on G, G_Q denote the query graph of a query Q and $r(G_Q)$ denote the radius of the query graph G_Q. Q can be evaluated using intra-partition processing over $\{P_1^k, P_2^k, \ldots, P_m^k\}$ if $r(G_Q) \leq k$.

We give a brief sketch of proof. By the k-hop forward (or reverse or bidirection) semantic hash partitioning, for any anchor vertex u in baseline partition P_i, all triples that are within k hops from u along the forward direction (or reverse or bidirection) are included in P_i^k. Therefore, it is guaranteed that all required triples to evaluate Q from u reside in the expanded partition if $r(G_Q) \leq k$.

2.4.3.3 Selective k-hop Expansion

Instead of replicating triples by expanding k hops in an exhaustive manner, we promote to further control the k-hop expansion by using some context-aware filters. For example, we can filter out some `rdf:type`-like triples that are rarely used in most of queries in the k-hop *reverse* expansion step to reduce the total number of triples to be replicated, based on the two observations. First, `rdf:type` predicate is widely used in most of RDF datasets to represent membership (or class) information of resources. Second, there are few object-object joins where more than one `rdf:type`-like triples are connected by an object variable, such as {Greg type ?x. Brian type ?x .}. By identifying such type of uncommon case, we can set a triple filter that will not replicate those `rdf:type`-like triples if their *object* vertices are the border vertices of the partition. However, we keep the `rdf:type`-like triples when performing forward direction expansion (i.e., from subject to object), because those triples are essential to provide fast pruning of irrelevant results due to the fact that the `rdf:type`-like triples in the forward direction typically are given as query conditions for most SPARQL queries. Our experimental results in Section 2.6 display significant reduction of replicated triples compared to the k-hop semantic hash partitioning without the object-based `rdf:type`-like triple filter.

2.4.3.4 URI Hierarchy-based Optimization

In an RDF graph, URI (Uniform Resource Identifier) references are used to identify vertices (except literals and blank nodes) and edges. URI references usually have a path hierarchy, and URI references having a common ancestor are often connected together, presenting high access locality. We conjecture that if such URI references (vertices) are placed in the same partition, we may reduce the number of replicated triples because a good portion of triples that need to be replicated by k-hop expansion from a vertex v are already located in the same partition of v. For example, the most common form of URI references in RDF datasets are URLs (Uniform Resource Locators) with http as their schema, such as "http://www.Department1.University2.edu/FullProfessor2/Publication14". The typical structure of URLs is "$http://domainname/path1/path2/\ldots/pathN\#fragmentID$". We

first extract the hierarchy of the domain name based on its levels and then add the path components and the fragment ID by keeping their order in the full URL path. For instance, the hierarchy of the previous example URL, starting from the top level, will be "edu", "University2", "Department1", "FullProfessor2", "Publication14". Based on this hierarchy, we measure the percentage of RDF triples whose subject vertex and object vertex share the same ancestor for different levels of the hierarchy. If, at any level of the hierarchy, the percentage of such triples is larger than a system-supplied threshold (empirically defined) and the number of distinct URLs sharing this common hierarchical structure is greater than or equal to the number of partition servers, we can use the selected portion of the hierarchy from the top to the chosen level, instead of full URI references, to participate in the baseline hash partitioning process. This is because the URI hierarchy-based optimization can increase the access locality of baseline hash partitions by placing triples whose subjects are sharing the same prefix structure of URLs into the same partitions, while distributing the large collection of RDF triples across all partition servers in a balanced manner. We call such preprocessing *the URI hierarchy optimization*.

In summary, when using a hash function to build the baseline partitions, we calculate the hash value on the selected part of URI references and place those triples having the same hash value on the selected part of URI references in the same partition. Our experiments reported in Section 2.6 show that with the URI hierarchy optimization, we can obtain a significant reduction of replicated triples at the k-hop expansion phase.

2.4.3.5 Algorithm and Implementation

Algorithm 1 shows the pseudocode for our semantic hash partitioning scheme. It includes the configuration of parameters at the initialization step and the k-hop semantic hash partitioning, which carries out in multiple Hadoop jobs. The first Hadoop job will perform two tasks: generating triple groups and generating baseline partitions by hashing anchor vertices of triple groups. The subsequent Hadoop job will generate k-hop semantic hash partitions ($k \geq 2$).

We assume that the input RDF graph has loaded into HDFS. The map function of the

Algorithm 1 Semantic Hash Partitioning

```
    Input: an RDF graph G, k, type (s-TG, o-TG or so-TG), direction (forward, reverse or bidirection)
    Output: a set of semantic hash partitions
 1: Initially, semantic partitions are empty
 2: Initially, there is no (anchor, border) pair
    Round=1 // generating baseline partitions
    Map
    Input: triple t(s,p,o)
 3: switch type do
 4:     case s − TG: emit(s,t)
 5:     case o − TG: emit(o,t)
 6:     case so − TG: emit(s,t), emit(o,t)
 7: end switch
    Reduce
    Input: key: anchor vertex anchor, value: triples
 8: add (hash(anchor), triples)
 9: if k = 1 then
10:     output baseline partitions P_1, ..., P_n
11: else
12:     read triples
13:     emit (anchor, borderSet)
14:     Round = Round + 1
15: end if
16: while Round ≤ k do //start k-hop triple replication
        Map
        Input: (anchor, border) pair or triple t(s,p,o)
17:     if (anchor, border) pair is read then
18:         emit(border, anchor)
19:     else
20:         switch direction do
21:             case forward: emit(s,t)
22:             case reverse: emit(o,t)
23:             case bidirection: emit(s,t), emit(o,t)
24:         end switch
25:     end if
        Reduce
        Input: key: border vertex border, value: anchors and triples
26:     for each anchor in anchors do
27:         add (hash(anchor), triples)
28:         if k < Round then
29:             read triples
30:             emit (anchor, borderSet)
31:         end if
32:     end for
33:     if k = Round then
34:         output semantic partitions P_1^k, ..., P_n^k
35:     end if
36:     Round = Round + 1
37: end while
```

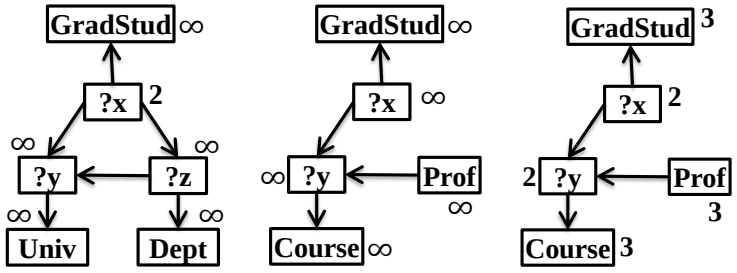
first Hadoop job reads each triple and emits a key-value pair in which the key is subject (for *s-TG*) or object (for *o-TG*) of the triple and the value is the remaining part of the triple. If we use *so-TG* for generating baseline partitions, the map function emits two key-value pairs, one using its subject as the key and the other using its object as the key (line 3-7). Next we generate triple groups based on the subject (or object or both subject and object) during the shuffling phase such that triples with the same anchor vertex are grouped together and assigned to the partition indexed by the hash value of their anchor

vertex. The reduce function records the assigned partition of the grouped triples using the hash value of their anchor vertex (line 8). If $k = 1$, we simply output the set of semantic hash partitions by merging all triples assigned to the same partition. Otherwise, the reduce function also records, for each anchor vertex, a set of vertices which should be expanded in the next hop expansion (line 9-15). We call such vertices *border vertices* of the anchor vertex. Concretely, for each triple in the triple group associated with the anchor vertex, the reduce function records the other vertex (e.g., the object vertex if the anchor vertex is the subject) as a border vertex of the anchor vertex because triples anchored at the border vertex may be selected for expansion in the next hop.

In the next Hadoop job, we implement k-hop semantic hash partitioning by controlled triple replication along the given expansion direction. The map function examines each baseline partition and reads a (anchor vertex, border vertex) pair, and emits a key-value pair in which the key is the border vertex and the value is the anchor vertex (line 17-25). During the shuffling phase, a set of anchor vertices that have the same border vertex are grouped together. The reduce function adds the triples connecting the border vertex to the partition if they are new to the partition and records the partition index of the triple using the hash value of the anchor vertex (line 27). If $k = 2$, we output the set of semantic partitions obtained so far. Otherwise, we record a set of new border vertices for each anchor vertex and repeat this job until k-hop semantic hash partitions are generated (line 28-31).

2.5 Distributed Query Processing

The distributed query processing component consists of three main tasks: query analysis, query decomposition, and distributed query execution. The query analyzer determines whether or not a query Q can be executed using intra-partition processing. All queries that can be evaluated by intra-partition processing will be sent to the distributed query plan execution module. For those queries that require inter-partition processing, the query decomposer is invoked to split Q into a set of subqueries, each can be evaluated by intra-partition processing. The distributed query execution planner will coordinate the joining of intermediate results from executions of subqueries to produce the final result of the query.



(a) Q1: forward (b) Q2: forward (c) Q2: bi-direction

Figure 5: Calculating Query Radius

2.5.1 Query Analysis

Given a query Q and its query graph, we first examine whether the query can be executed using intra-partition processing. According to Theorem 1, we calculate the *radius* and the center vertices of the query graph based on Definition 8, denoted by $r(Q)$ and $center(Q)$ respectively. If the dataset is partitioned using the k-hop expansion, then we evaluate whether $r(Q) \leq k$ holds. If yes, the query Q as a whole can be executed using the intra-partition processing. Otherwise, the query Q is passed to the query decomposer.

Fig. 5 presents three example queries with their query graphs respectively. We place the eccentricity value of each vertex next to the vertex. Since the *forward* radius of the query graph in Fig. 5(a) is 2, we can execute the query using intra-partition processing if the query is issued against the k-hop forward semantic hash partitions and k is equal to or larger than 2. In Fig. 5(b), the *forward* radius of the query graph is *infinity* because there is no vertex which has at least one forward direction path to all other vertices. Therefore, we cannot execute the query over the k-hop forward semantic hash partitions using intra-partition processing regardless of the hop count value of k. This query is passed to the query decomposer for further query analysis. Fig. 5(c) shows the eccentricity of vertices in the query graph under the *bidirection* semantic hash partitions. The *bidirection* radius is 2 and there are two center vertices: `?x` and `?y`. Therefore we can execute the query using intra-partition processing if k is equal to or larger than 2 under the bidirection semantic hash partitions.

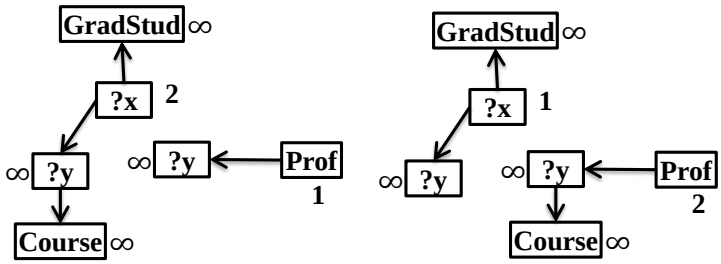


Figure 6: Query Decomposition

2.5.2 Query Decomposition

The first issue in evaluating a query Q using *inter-partition* processing is to determine the number of subqueries Q needs to be decomposed into. Given that there are more than one way to split Q into a set of subqueries, an intuitive approach is to first check whether Q can be decomposed into *two* subqueries such that each subquery can be evaluated using intra-partition processing. If there is no such decomposition, then we increase the number of subqueries by one and check again to see whether the decomposition enables each subquery to be evaluated by intra-partition processing. We repeat this process until a desirable decomposition is found.

Concretely, we start the query decomposition by putting all vertices in the query graph of Q into a set of candidate vertices to be examined in order to find such a decomposition having two subqueries. For each candidate vertex v, we find the largest subgraph from v, in the query graph of Q, which can be executed using intra-partition processing under the current k-hop semantic hash partitions. For the remaining part of the query graph, which is not covered by the subgraph, we check whether there is any vertex whose expanded subgraph under the current k-hop expansion can fully cover the remaining part. If there is such a decomposition, we treat each subgraph as a subquery of Q. Otherwise, we increase the number of subqueries by one and then repeat the above process until we find a possible decomposition. If we find several possible decompositions having the equal number of subqueries, then we choose the one in which the standard deviation of the size (i.e., the number of triple patterns) of subqueries is the smallest, under the assumption that a small

Algorithm 2 Join Processing

```
Input: two intermediate results, join variable list, output variable list
Output: joined results
Map
Input: one tuple from one of the two intermediate results
1: Extracts a list of values, from the tuple, which are corresponding to the join variables
2: emit(join values, the remaining values of the tuple)
Reduce
Input: key: join values, value: two sets of tuples
3: Generates the Cartesian product of the two sets
4: Projects only columns that are included in the output variables
5: return joined (and projected) results
```

subquery may generate large intermediate results. We leave as future work the query optimization problem where we can utilize additional metadata such as query selectivity information.

For example, in Fig. 5(b) where the query cannot be executed using intra-partition processing under the *forward* semantic hash partitions, assume that partitions are generated using the 2-hop forward direction expansion. To decompose the query, if we start with vertex `?x`, we will get a decomposition that consists of two subqueries as shown in Fig. 6(a). If we start with vertex `Prof`, we will also get two subqueries as shown in Fig. 6(b). Based on the smallest subquery standard deviation criterion outlined above, we choose the latter because two subqueries are of the same size.

2.5.3 Distributed Query Execution

Intra-partition processing steps: Let the number of partition servers be N. If the query Q can be executed using *intra-partition* processing, we send Q to each of the N partition servers in parallel. Upon the completion of local query execution, each partition server will send the partial results generated locally to the master server, which merges the results from all partition servers to generate the final results. The entire processing does not involve any coordination and communication among partition servers. The only communication happens between the master server and all its slave servers to ship the query to all slave servers and ship partial results from slaves to the master server.

Inter-partition processing steps: If the query Q cannot be executed using intra-partition processing, the query decomposer will be invoked to split Q into a set of subqueries. Each

subquery is executed in all partitions using intra-partition processing and then the intermediate results of all sub-queries are loaded into HDFS and joined using Hadoop MapReduce. To join the two intermediate results, the map function of a Hadoop job reads each tuple from the two results and extracts a list of values, from the tuple, which are corresponding to the join variables. Then the map function emits a key-value pair in which the key is the list of extracted values (i.e., join key) and the value is the remaining part of the tuple. Through the shuffling phase of MapReduce, two sets of tuples sharing the same join values are grouped together: one is from the first intermediate results and the other is from the second intermediate results. The reduce function of the job generates the Cartesian product of the two sets and projects only columns that are included in the output variables or will be used in subsequent joins. Finally, the reduce function records the projected tuples. Algorithm 2 shows the pseudocode for our join processing during inter-partition processing. Since we use one Hadoop job to join the intermediate results of two subqueries, more subqueries usually imply more query processing and higher query latency due to the large overhead of Hadoop jobs.

2.6 Experimental Evaluation

This section reports the experimental evaluation of our semantic hash partitioning scheme using our prototype system SHAPE. We divide the experimental results into four sets: (i) We present the experimental results on loading time, redundancy, and triple distribution. (ii) We conduct the experiments on query processing latency, showing that by combining the semantic hash partitioning with the intra-partition processing-aware query partitioning, our approach reduces the query processing latency considerably compared to existing simple hash partitioning and graph partitioning schemes. (iii) We also evaluate the scalability of our approach with respect to varying dataset sizes and varying cluster sizes. (iv) We also evaluate the effectiveness of our optimization techniques used for reducing the partition size and the amount of triple replication.

2.6.1 Experimental Setup and Datasets

We use a cluster of 21 physical servers (one master server) on Emulab [115]: each has 12 GB RAM, one 2.4 GHz 64-bit quad core Xeon E5530 processor, and two 250GB 7200 rpm SATA disks. The network bandwidth is about 40 MB/s. When we measure the query processing time, we perform five cold runs under the same setting and show the *fastest* time to remove any possible bias posed by OS and/or network activity. We use RDF-3X version 0.3.5, installed on each slave server. We use Hadoop version 1.0.4 running on Java 1.6.0 to run various partitioning algorithms and join the intermediate results generated by subqueries.

We experiment with our 2-hop forward (*2f*), 3-hop forward (*3f*), 4-hop forward (*4f*), 2-hop bidirection (*2b*), and 3-hop bidirection (*3b*) semantic hash partitions, with the `rdf:type`-like triple optimization and the URI hierarchy optimization, expanded from the baseline partitions on subject-based triple groups. To compare our semantic hash partitions, we have implemented the random partitioning (*rand*), the simple hash partitioning on subjects (*hash-s*), the simple hash partitioning on both subjects and objects (*hash-so*), and the graph partitioning [55] with undirected 2-hop guarantee (*graph*). For fair comparison, we apply the `rdf:type`-like triple optimization to *graph*.

To run the vertex partitioning of *graph*, we also use the graph partitioner METIS [15] version 5.0.2 with its default configuration. We do not directly compare with other partitioning techniques that do not use the RDF-specific storage system to store RDF triples, such as SHARD [97], because it is reported in [55] that they are much slower than the graph partitioning for all benchmark queries. The random partitioning (*rand*) is similar to using HDFS for partitioning, but more optimized in the storage level by using the RDF-specific storage system.

For our evaluation, we use eight datasets of different sizes from four domains as shown in Table. 1. **LUBM** [47] and **SP^{2}Bench** [101] are benchmark generators and **DBLP** [1], containing bibliographic information in computer science, and **Freebase** [5], a large knowledge base, are the two real RDF datasets. As a data cleaning step, we remove any duplicate triples using one Hadoop job.

Table 1: Datasets (SHAPE)

Dataset	#Triples	#subjects	#rdf:type triples
LUBM267M	267M	43M	46M
LUBM534M	534M	87M	92M
LUBM1068M	1068M	174M	184M
SP2B200M	200M	36M	36M
SP2B500M	500M	94M	94M
SP2B1000M	1000M	190M	190M
DBLP	57M	3M	6M
Freebase	101M	23M	8M

2.6.2 Data Loading Time

Table 2 shows the data loading time of the datasets for different partitioning algorithms. Due to the space limit, we report the results of the largest dataset among three benchmark datasets. The data loading time basically consists of the data partitioning time and the partition loading time into RDF-3X. For *graph*, one additional step is required to run METIS for vertex partitioning. Note that the graph partitioning approach using METIS fails to work on large datasets, such as LUBM534M, LUBM1068M, SP2B500M, and SP2B1000M, due to the insufficient memory. The random partitioning (*rand*) and the simple hash partitioning on subjects (*hash-s*) have the fastest loading time because they just need to read each triple and assign the triple to a partition randomly (*rand*) or based on the hash value of the triple's subject (*hash-s*). Our forward direction-based approaches have fast loading time. The graph partitioning (*graph*) has the longest loading time if METIS can process the input dataset. For example, it takes about 25 hours to convert the Freebase dataset to a METIS input format and about 44 minutes to run METIS on the input. Note that our converter (from RDF to METIS input format), implemented using Hadoop MapReduce, is not the problem of this slow conversion time because, for LUBM267M, it takes 38 minutes (33 minutes for conversion and 5 minutes for running METIS), much faster than the reported time (1 hour) in [55].

2.6.3 Redundancy and Triple Distribution

Table 3 shows, for each partitioning algorithm, the ratio of the number of triples in all generated partitions to the total number of triples in the original datasets. The random partitioning (*rand*) and the simple hash partitioning on subjects (*hash-s*) have the ratio of

Table 2: Partitioning and Loading Time (min)

Algorithm	METIS	Partitioning	Loading	Total
LUBM1068M				
single server	-	-	779	779
rand	-	17	47	64
hash-s	-	19	34	53
hash-so	-	84	131	215
graph	fail	N/A	N/A	N/A
2-forward	-	94	32	126
3-forward	-	117	32	149
4-forward	-	133	32	165
2-bidirection	-	121	61	182
3-bidirection	-	396	554	950
SP2B1000M				
single server	-	-	665	665
rand	-	16	39	55
hash-s	-	16	28	44
hash-so	-	74	81	155
graph	fail	N/A	N/A	N/A
2-forward	-	89	34	123
3-forward	-	111	34	145
4-forward	-	127	34	161
2-bidirection	-	109	53	162
3-bidirection	-	195	135	330
Freebase				
single server	-	-	73	73
rand	-	2	4	6
hash-s	-	2	3	5
hash-so	-	5	9	14
graph	1573	38	52	1663
2-forward	-	9	4	13
3-forward	-	11	4	15
4-forward	-	14	4	18
2-bidirection	-	22	17	39
3-bidirection	-	59	75	134
DBLP				
single server	-	-	34	34
rand	-	2	2	4
hash-s	-	2	1	3
hash-so	-	4	3	7
graph	452	22	35	509
2-forward	-	7	2	9
3-forward	-	8	2	10
4-forward	-	10	2	12
2-bidirection	-	13	8	21
3-bidirection	-	36	35	71

1 because there is no replicated triple. This result shows that our forward direction-based approaches can reduce the number of replicated triples considerably while maintaining the hop guarantee. For example, even though we expand the baseline partitions to satisfy 4-hop guarantee (forward direction), the replication ratio is less than 1.6 for all the datasets. On the other hand, this result also shows that we should be careful when we expand the baseline partitions using both directions. Since the original data can be almost fully replicated on all the partitions when we use 3-hop bidirection expansion, the number of hops should be

decided carefully by considering the tradeoff between the overhead of local processing and inter-partition communication. We leave how to find an optimal k value, given a dataset and a set of queries, as future work.

Table 3: Redundancy (Ratio to Original Dataset)

Dataset	2f	3f	4f	2b	3b	hash-so	graph
LUBM267M	1.00	1.00	1.00	1.67	8.87	1.78	3.39
LUBM534M	1.00	1.00	1.00	1.67	8.73	1.78	N/A
LUBM1068M	1.00	1.00	1.00	1.67	8.66	1.78	N/A
SP2B200M	1.18	1.19	1.19	1.76	3.81	1.78	1.32
SP2B500M	1.16	1.17	1.17	1.70	3.58	1.77	N/A
SP2B1000M	1.15	1.15	1.16	1.69	3.50	1.77	N/A
DBLP	1.48	1.53	1.55	5.35	18.28	1.86	5.96
Freebase	1.18	1.26	1.28	5.33	17.18	1.87	7.75

Table 4 shows the coefficient of variation (the ratio of the standard deviation to the mean) of generated partitions in terms of the number of triples to measure the dispersion of the partitions. Having uniformly distributed triples across all partitions is one of the key performance factors because the large partitions in the skewed distribution can be performance bottlenecks during query processing. Our semantic hash partitioning approaches have almost perfect uniform distributions. On the other hand, the results indicate that partitions generated using *graph* are very different in size. For example, among the partitions generated using *graph* for DBLP, the largest partition is 3.8 times bigger than the smallest partition.

Table 4: Distribution (Coefficient of Variation)

Dataset	2f	3f	4f	2b	3b	hash-so	graph
LUBM267M	0.01	0.01	0.01	0.00	0.01	0.20	0.26
LUBM534M	0.01	0.01	0.01	0.00	0.01	0.20	N/A
LUBM1068M	0.01	0.01	0.01	0.00	0.01	0.20	N/A
SP2B200M	0.00	0.00	0.00	0.00	0.00	0.01	0.05
SP2B500M	0.00	0.00	0.00	0.00	0.00	0.01	N/A
SP2B1000M	0.00	0.00	0.00	0.00	0.00	0.01	N/A
DBLP	0.00	0.00	0.00	0.00	0.00	0.09	0.50
Freebase	0.00	0.00	0.00	0.00	0.00	0.16	0.24

2.6.4 Query Processing

For our query evaluation of the three LUBM datasets, we report the results of all 14 benchmark queries provided by **LUBM**. Among the 14 queries, 8 queries (Q1, Q3, Q4, Q5, Q6, Q10, Q13, and Q14) are star queries. The *forward* radii of Q2, Q7, Q8, Q9, Q11, and Q12 are 2, ∞, 2, 2, 2, and 2 respectively. Their *bidirection* radii are all 2. Due to the space

limit, for the other datasets, we report the results of one star query and one or two complex queries including chain-like patterns. We pick three queries among a set of benchmark queries provided by **SP^{2}Bench** and create star and complex queries for the real datasets. Table 5 shows the queries used for our query evaluation. The *forward* radii of SP2B Complex1 and Complex2 are ∞ and 2 respectively. The *bidirection* radii of SP2B Complex1 and Complex2 are 3 and 2 respectively.

Table 5: SPARQL Queries

LUBM	**All 14** benchmark queries
SP2B Star	Benchmark Query2 (without Order by and Optional) Select ?inproc ?author ?booktitle ?title ?proc ?ee ?page ?url ?yr Where { ?inproc rdf:type Inproceedings . ?inproc creator ?author . ?inproc booktitle ?booktitle . ?inproc title ?title . ?inproc partOf ?proc . ?inproc seeAlso ?ee . ?inproc pages ?page . ?inproc homepage ?url . ?inproc issued ?yr }
SP2B Complex1	Benchmark Query4 (without Filter) Select DISTINCT ?name1 ?name2 Where { ?article1 rdf:type Article . ?article2 rdf:type Article . ?article1 creator ?author1 . ?author1 name ?name1 . ?article2 creator ?author2 . ?author2 name ?name2 . ?article1 journal ?journal . ?article2 journal ?journal }
SP2B Complex2	Benchmark Query6 (without Optional) Select ?yr ?name ?document Where { ?class subClassOf Document . ?document rdf:type ?class . ?document issued ?yr . ?document creator ?author . ?author name ?name }
DBLP Star	Select ?author ?name Where { ?author rdf:type Agent . ?author name ?name }
DBLP Complex	Select ?paper ?conf ?editor Where { ?paper partOf ?conf . ?conf editor ?editor . ?paper creator ?editor }
Freebase Star	Select ?person ?name Where { ?person gender male . ?person rdf:type book.author . ?person rdf:type people.person . ?person name ?name }
Freebase Complex1	Select ?loc1 ?loc2 ?postal Where { ?loc1 headquarters ?loc2 . ?loc2 postalcode ?postal . }
Freebase Complex2	Select ?name1 ?name2 ?birthplace ?inst Where { ?person1 birth ?birthplace . ?person2 birth ?birthplace . ?person1 education ?edu1 . ?edu1 institution ?inst . ?person2 education ?edu2 . ?edu2 institution ?inst . ?person1 name ?name1 . ?person2 name ?name2 . ?edu1 rdf:type education . ?edu2 rdf:type education }

Fig. 7 shows the query processing time of all 14 benchmark queries for different partitioning approaches on LUBM534M dataset. Since the results of our 2-hop forward (*2f*), 3-hop forward (*3f*), and 4-hop forward (*4f*) partitions are almost the same, we merge them into one. Our forward direction-based partitioning approaches (*2f*, *3f*, and *4f*) have faster query processing time than the other partitioning techniques for all the benchmark queries except Q7 in which inter-partition processing is required for *2f*, *3f*, and *4f*. Our 2-hop bidirection

Table 6: Query Processing Time (sec)

Dataset	2f	3f	4f	2b	3b	hash-s	hash-so	random	graph	single server
SP2B200M Star	64.37	68.00	71.01	69.08	344.70	**58.56**	129.42	835.00	73.93	500.83
SP2B200M Complex1	**222.96**	224.03	225.32	226.98	659.77	1257.99	2763.82	2323.42	223.60	fail
SP2B200M Complex	**42.70**	53.13	56.39	52.39	136.14	208.38	357.72	341.83	49.79	431.44
SP2B500M Star	183.41	187.02	197.72	197.61	967.29	**166.92**	378.61	1625.76	N/A	fail
SP2B500M Complex1	**487.08**	493.48	522.40	529.83	1272.77	3365.15	7215.62	5613.11	N/A	fail
SP2B500M Complex2	**113.00**	115.27	116.96	125.74	410.66	449.63	921.24	703.55	N/A	1690
SP2B1000M Star	456.12	479.47	482.59	459.58	2142.45	**413.24**	685.49	2925.02	N/A	fail
SP2B1000M Complex1	**897.88**	911.01	917.83	1006.93	2391.88	6418.56	14682	11740	N/A	fail
SP2B1000M Complex2	**258.61**	265.82	270.21	282.41	905.22	808.23	1986.95	1353.03	N/A	fail
DBLP Star	12.88	12.91	13.59	17.94	41.18	**3.62**	5.33	56.01	30.51	22.71
DBLP Complex	**3.48**	3.57	3.73	9.90	31.66	61.28	74.19	117.48	20.87	21.38
Freebase Star	10.67	11.15	11.95	22.06	129.11	**8.23**	9.02	143.60	105.41	42.61
Freebase Complex1	**6.99**	7.78	8.07	13.68	71.44	54.36	61.78	57.54	25.41	43.59
Freebase Complex2	**63.80**	66.87	67.28	80.48	501.56	216.56	238.57	568.92	195.98	23213

(*2b*) approach also has good performance because it ensures intra-partition processing for all benchmark queries.

For Q7, since our forward direction-based partitioning approaches need to run one Hadoop job to join the intermediate results of two subqueries and the size of the intermediate results is about 2.4 GB (much larger compared to the final result size of 907 bytes), its query processing time for Q7 is very slow compared to other approaches (*2b* and *3b*) using intra-partition processing. However, our approaches (*2f*, *3f*, and *4f*) are faster than the simple hash partitioning (*hash-s* and *hash-so*), which requires two Hadoop jobs to process Q7. Recall that the graph partitioning does not work for LUBM534M because METIS failed due to the insufficient memory.

Table 6 shows the query processing times of the other datasets. The fastest query processing time for each query is marked in bold. Our forward direction-based partitioning approaches (*2f*, *3f*, and *4f*) are faster than the other partitioning techniques for all complex queries. For example, for SP2B1000M Complex1, our approach *2f* is about 7, 16,

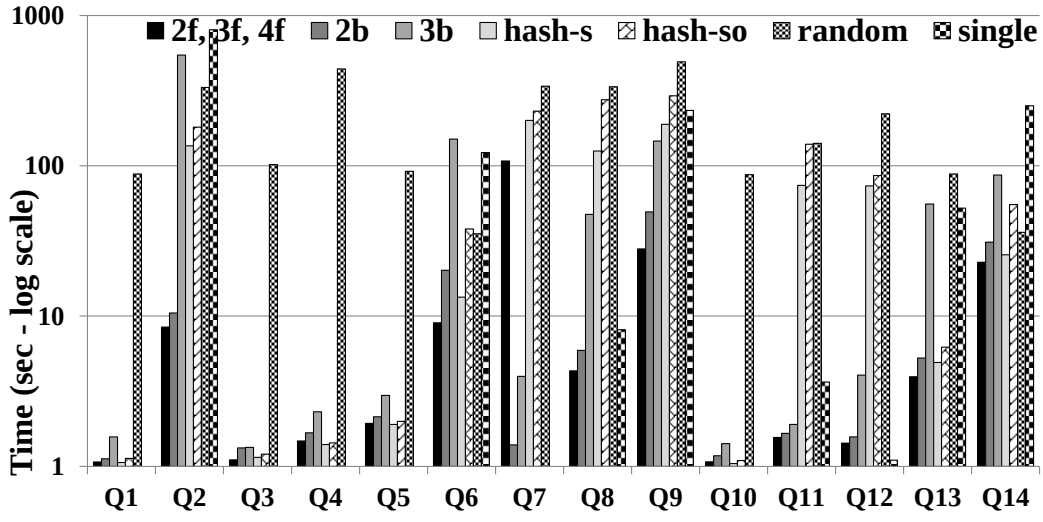
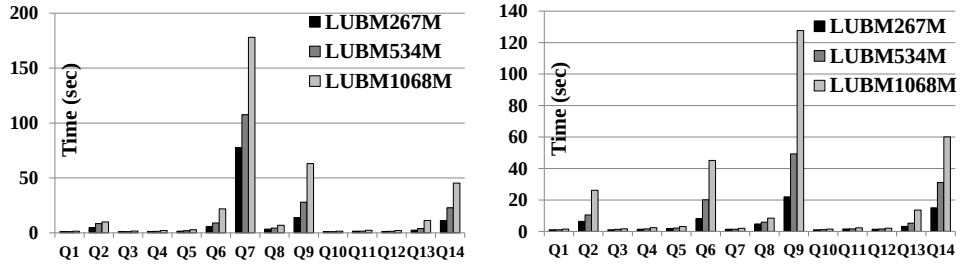


Figure 7: Query Processing Time (LUBM534M)



(a) 2-hop forward (2f)

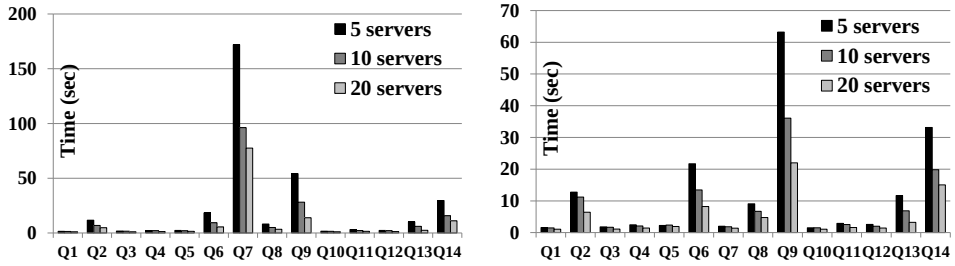
(b) 2-hop bidirection (2b)

Figure 8: Scalability with Varying Dataset Sizes

and 13 times faster than *hash-s*, *hash-so*, and *random* respectively. Note that executing SP2B1000M Complex1 fails on a single server due to the insufficient memory and *graph* does not work for SP2B1000M. Our 2-hop bidirection (*2b*) approach also has comparable query processing performance with *2f*, *3f*, and *4f*. Even though our 3-hop bidirection (*3b*) approach is much slower than *2f*, *3f*, *4f*, and *2b* due to its large partition size, it is faster than *random* for most queries. For star queries, *hash-s* is slightly faster than our approaches because it is optimized only for star queries and there is no replicated triple.

2.6.5 Scalability

We evaluate the scalability of our partitioning approach by varying dataset sizes and cluster sizes. Fig. 8 shows that the increase of the query processing time of star queries Q6, Q13, and Q14 is almost proportional to the dataset size. For Q7, under the 2-hop bidirection



(a) 2-hop forward (2f) (b) 2-hop bidirection (2b)

Figure 9: Scalability with Varying Cluster Sizes

(*2b*) expansion, the query processing time increases only slightly because its results do not change by the dataset size. On the other hand, under the 2-hop forward (*2f*) expansion, there is a considerable increase in the query processing time because the intermediate results increase according to the dataset size even though the final results are the same regardless of the dataset size.

Fig. 9 shows the results of scalability experiment with varying numbers of slave servers from 5 to 20 on LUBM267M dataset. For star queries whose selectivity is high (Q1, Q3, Q4, Q5, and Q10), the processing time slightly decreases with an increasing number of servers due to the reduced partition size. For star queries with low selectivity (Q6, Q13, and Q14), the decrease of the query processing time is almost proportional to the number of slave servers.

2.6.6 Effects of Optimizations

Table 7 shows the effects of different optimization techniques under the 2-hop bidirection (*2b*) expansion in terms of the replication ratio. Without any optimization, large partitions are generated because lots of triples are replicated and so it will considerably increase the query processing time. Using the `rdf:type`-like triple optimization, we can reduce the partition size by excluding `rdf:type`-like triples during the expansion. The result of applying the URI hierarchy optimization shows that we place close vertices in the same partition and so prevent the replication of many triples. By combining both optimization techniques, we substantially reduce the partition size and so increase the performance of query processing.

Table 7: Effects of Optimizations (Replication Ratio)

Dataset	No Opt.	rdf:type	URI hierarchy	Both
LUBM1068M	11.46	8.46	4.94	1.67
SP2B1000M	6.95	3.70	5.12	1.69
DBLP	7.24	5.35	N/A	5.35
Freebase	6.88	5.33	N/A	5.33

2.7 Conclusion

In this chapter we have shown that when data needs to be partitioned across multiple server nodes, the choice of data partitioning algorithms can make a big difference in terms of the cost of data shipping across a network of servers. We have presented a novel semantic hash partitioning approach, which starts with the simple hash partitioning and expands each partition by replicating only necessary triples to increase access locality and promote intra-partition processing of SPARQL queries. We have also developed a partition-aware distributed query processing facility to generate locality-optimized query execution plans. In addition, we have provided a suite of locality-aware optimization techniques to further reduce the partition size and cut down on the inter-partition communication cost during distributed query processing. Our experimental results show that the semantic hash partitioning approach improves the query latency and is more efficient than existing popular simple hash partitioning and graph partitioning schemes.

The first prototype system for our semantic hash partitioning does not support aggregate queries and update operations. We plan to implement new features introduced in SPARQL 1.1. Both rdf:type filter and URI hierarchy-based merging of triple groups are provided as a configuration parameter. One of our future work is to utilize statistics collected over representative set of queries to derive a near-optimal setting of k for k-hop semantic hash partitioning. Finally, we conjecture that the effectiveness of RDF data partitioning can be further enhanced by exploring different strategies for access locality-guided triple grouping and triple replication.

CHAPTER III

VB-PARTITIONER: EFFICIENT DATA PARTITIONING FRAMEWORK FOR HETEROGENEOUS GRAPHS

As the size and variety of information networks continue to grow in many scientific and engineering domains, we witness a growing demand for efficient processing of large heterogeneous graphs using a cluster of compute nodes in the Cloud. One open issue is how to effectively partition a large graph to process complex graph operations efficiently. In this chapter, we present VB-Partitioner – a distributed data partitioning model and algorithms for efficient processing of graph operations over large-scale graphs in the Cloud. Our VB-Partitioner has three salient features. First, it introduces vertex blocks (VBs) and extended vertex blocks (EVBs) as the building blocks for semantic partitioning of large graphs. Second, VB-Partitioner utilizes vertex block grouping algorithms to place those vertex blocks that have high correlation in graph structure into the same partition. Third, VB-Partitioner employs a VB-partition guided query partitioning model to speed up the parallel processing of graph pattern queries by reducing the amount of inter-partition query processing. We conduct extensive experiments on several real-world graphs with millions of vertices and billions of edges. Our results show that VB-Partitioner significantly outperforms the popular random block-based data partitioner in terms of query latency and scalability over large-scale graphs.

3.1 Introduction

Many real-world information networks consist of millions of vertices representing heterogeneous entities and billions of edges representing heterogeneous types of relationships among entities, such as Web-based networks, social networks, supply-chain networks, and biological networks. One concrete example is the phylogenetic forests of bacteria, where each node represents a genetic strain of Mycobacterium tuberculosis complex (MTBC) and each edge represents a putative evolutionary change. Processing large heterogeneous graphs poses

a number of unique characteristics in terms of big data processing. First, graph data are highly correlated, and the topological structure of a big graph can be viewed as a correlation graph of its vertices and edges. Heterogeneous graphs add additional complexity compared to homogeneous graphs in terms of both storage and computation due to the heterogeneous types of entity vertices and entity links. Second, queries over graphs are typically subgraph matching operations. Thus, we argue that modeling heterogeneous graphs as a big table of entity vertices or entity links is ineffective for parallel processing of big graphs in terms of storage, network I/O, and computation.

Hadoop MapReduce programming model and Hadoop Distributed File System (HDFS) are among the most popular distributed computing technologies for partitioning big data processing across a large cluster of compute nodes in the Cloud. HDFS (and its attached storage systems) is excellent for managing the big table data when row objects are independent and thus big data can be simply divided into equal-sized blocks (chunks) that can be stored and processed in parallel efficiently and reliably. However, HDFS is not optimized for storing and partitioning big datasets of high correlation, such as large graphs [80, 67]. This is because HDFS's block-based partitioning is equivalent to random partitioning of big graph data through either horizontal vertex-based partitioning or edge-based partitioning depending on whether the graph is stored physically by entity vertices or by entity links. Therefore, data partitions generated by such a random partitioning method tend to incur unnecessarily large inter-partition processing overheads due to the high correlation and thus the need for high degree of interactions among partitions in responding to a graph pattern query. Using such random partitioning methods, even for simple graph pattern queries, may incur unnecessarily large inter-partition join processing overheads due to the high correlation among partitions and demand multiple rounds of data shipping across partitions stored in multiple nodes of a compute cluster in the Cloud. Thus, Hadoop MapReduce alone is neither adequate for handling graph pattern queries over large graphs nor suitable for structure-based reasoning on large graphs, such as finding k-hop neighbors satisfying certain semantic constraints.

In this chapter, we present a vertex block-based partitioning and grouping framework,

called VB-Partitioner, for scalable and yet customizable graph partitioning and distributed processing of graph pattern queries over big graphs. VB-Partitioner supports three types of vertex blocks and a suite of vertex block grouping strategies, aiming at maximizing the amount of local graph processing and minimizing the network I/O overhead of inter-partition communication during each graph processing job. We demonstrate the efficiency and effectiveness of our VB-Partitioner by developing a VB-partition guided computation partitioning model that allows us to decompose graph pattern queries into desired vertex block partitions that are efficient for parallel query processing using a compute cluster.

This chapter makes three novel contributions. First, we introduce vertex blocks and extended vertex blocks as the building blocks for partitioning a large graph. This vertex block-based approach provides a foundation for scalable and yet customizable data partitioning of large heterogeneous graphs by preserving the basic vertex structure. By scalable, we mean that data partitions generated by VB-Partitioner can support fast processing of big graph data of different size and complexity. By customizable, we mean that one partitioning technique may not fit all. Thus, VB-Partitioner supports three types of vertex blocks and is by design adaptive to different data processing demands in terms of explicit and implicit structural correlations. Second, we develop a suite of vertex block grouping algorithms that enable efficient grouping of those vertex blocks with high correlation in graph structure into one *VB* partition. We optimize the vertex block grouping quality by maximizing the amount of local graph processing and minimizing the inter-partition communication during each graph processing job. Third, to further utilize our vertex block-based graph partitioning approach, we introduce a *VB-partition* guided computation partitioning model, which allows us to transform graph pattern queries into vertex block-based graph query patterns. By partitioning and distributing big graph data using vertex block-based partitions, powered by the *VB-partition* guided query partitioning model, we can considerably reduce the inter-node communication overhead for complex query processing because most graph pattern queries can be evaluated locally on a partition server without requiring data shipping from other partition nodes. We evaluate our data partitioning framework and

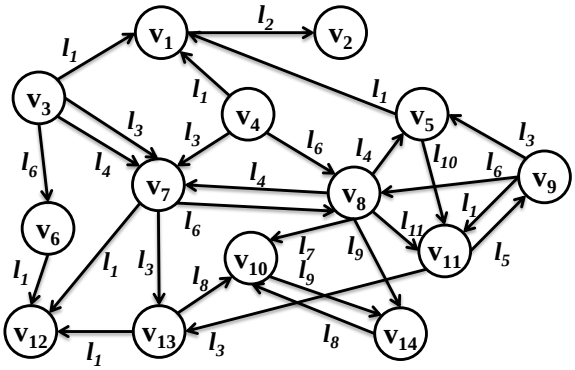


Figure 10: Heterogeneous Graph

algorithms through extensive experiments using both benchmark and real datasets with millions of vertices and billions of edges. Our experimental results show that VB-Partitioner is scalable and customizable for partitioning and distributing big graph datasets of diverse size and structures, and effective for processing real-time graph pattern queries of different types and complexity.

3.2 Overview

3.2.1 Heterogeneous Graphs

We first define the heterogeneous graphs as follows.

Definition 9 (Heterogeneous Graph) Let $\mathcal{V}$ be a countably infinite set of vertex names, and Σ_V and Σ_E be a finite set of available types (or labels) for vertices and edges respectively. A heterogeneous graph is a directed, labeled graph, denoted as $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$ where V is a set of vertices (a finite subset of $\mathcal{V}$) and E is a set of directed edges (i.e., $E \subseteq V \times \Sigma_E \times V$). In other words, we represent each edge as a triple (v, l, v') that is a l-labeled edge from v to v'. l_V is a map from a vertex to its type ($l_V : V \rightarrow \Sigma_V$) and l_E is a map from an edge to its label ($l_E : E \rightarrow \Sigma_E$).

Fig. 10 shows an example of heterogeneous graphs. For example, there are several l_1-labeled edges such as (v_3, l_1, v_1), (v_4, l_1, v_1) and (v_{13}, l_1, v_{12}). Homogeneous graphs are special cases of heterogeneous graphs where vertices are of the same type, such as Web pages, and edges are of the same type, such as page links in a Web graph. In a heterogeneous

graph, each vertex may have incoming edges (in-edges) and outgoing edges (out-edges). For example, in Fig. 10, vertex v_7 has 3 out-edges and 4 in-edge (i.e., 7 bi-edges).

Definition 10 (Out-edges, in-edges, and bi-edges) Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, the set of **out-edges** of a vertex $v \in V$ is denoted by $E_v^+ = \{(v, l, v') | (v, l, v') \in E\}$. Conversely, the set of **in-edges** of v is denoted by $E_v^- = \{(v', l, v) | (v', l, v) \in E\}$. We also define **bi-edges** of v as the union of its **out-edges** and **in-edges**, denoted by $E_v^\pm = E_v^+ \cup E_v^-$.

Definition 11 (Path) Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, an **out-edge path** from a vertex $u \in V$ to another vertex $w \in V$ is a sequence of vertices, denoted by $v_0, v_1, \ldots, v_k$, such that $v_0 = u$, $v_k = w$, $\forall m \in [0, k-1] : (v_m, l_m, v_{m+1}) \in E$. Conversely, an **in-edge path** from vertex u to vertex w is a sequence of vertices, denoted by $v_0, v_1, \ldots, v_k$, such that $u = v_0$, $w = v_k$, $\forall m \in [0, k-1] : (v_{m+1}, l_m, v_m) \in E$. A **bi-edge path** from vertex u to vertex w is a sequence of vertices, denoted by $v_0, v_1, \ldots, v_k$, such that $u = v_0$, $w = v_k$, $\forall m \in [0, k-1] : (v_m, l_m, v_{m+1}) \in E$ or $(v_{m+1}, l_m, v_m) \in E$. The **length** of the path $v_0, v_1, \ldots, v_k$ is k.

Definition 12 (Hop count) Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, the **out-edge hop count** from a vertex $u \in V$ to another vertex $w \in V$, denoted by $hop^+(u, w)$, is the minimum length of all possible *out-edge* paths from u to w. We also define the out-edge hop count from u to an out-edge (w, l, w') of w, denoted by $hop^+(u, wlw')$, as $hop^+(u, w) + 1$. The hop count $hop^+(u, w)$ is zero if $u = w$ and ∞ if there is no out-edge path from u to w.

The in-edge and bi-edge hop counts are similarly defined using the in-edge and bi-edge paths respectively.

3.2.2 Operations on Heterogeneous Graphs

Graph pattern queries [30] are subgraph matching problems and are widely recognized as one of the most fundamental graph operations. A graph pattern is often expressed in terms of a set of vertices and edges such that some of them are variables. Processing of a graph pattern query is to find a set of vertex or edge values on the input graph that can be substituted for the variables while satisfying the structure of the graph pattern. Therefore,

processing a graph pattern query can be viewed as solving a subgraph matching problem or finding missing vertex or edge instantiation values in the input graph.

A *basic graph pattern* is an edge (v, l, v') in which any combination of the three elements can be variables. We represent variables with a prefix "?" such as `?x` to differentiate variables from the instantiation of vertex names and edge labels.

A *graph pattern* consists of a set of basic graph patterns. If there is a variable shared by several basic graph patterns, the returned values for the variable should satisfy all the basic graph patterns, which include the variable. For example, a graph pattern $\{(?x, l_1, v_8)$, $(?x, l_6, ?a)$, $(?x, l_3, ?b)\}$ requests those vertices that have l_1-labeled out-edge to v_8 and also l_6-labeled and l_3-labeled out-edges. It also requests the connected vertices (i.e., $?a$ and $?b$) linked by the out-edges. This type of operations is very common in social networks when we request additional information of users satisfying a certain condition such as $\{(?member, affiliation, GT)$, $(?member, hometown, ?city)$, $(?member, birthday, ?date)\}$. Another graph pattern $\{(?x, l_3, ?z)$, $(?x, l_6, ?y)$, $(?z, l_4, ?y)$, $(?z, l_1, ?a)\}$ requests all vertices such that each vertex x has any l_3-labeled (to z) and l_6-labeled (to y) out-edges and there is any l_4-labeled edge from z to y and z has any l_1-labeled out-edge. This type of operations is also common in social networks when we want to find friends of friends within k-hops satisfying a certain condition. We formally define the graph pattern as follows.

Definition 13 (graph pattern) Let $\mathcal{V}_{var}$ and $\mathcal{E}_{var}$ be countably infinite sets of vertex variables and edge variables respectively. Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, a graph pattern is $G_q = (V_q, E_q, \Sigma_{V_q}, \Sigma_{E_q}, l_{V_q}, l_{E_q})$ where $V_q \subseteq V \cup \mathcal{V}_{var}$ and $E_q \subseteq V_q \times (\Sigma_E \cup \mathcal{E}_{var}) \times V_q$.

For example, $\{(?member, work, ?company)$, $(?member, friend, ?friend)$, $(?friend$, $work, ?company)$, $(?friend, friend, ?friend2)\}$ requests, for each user, friends of her friends who are working in the same company with her. Fig. 11 gives four typical graph pattern queries (selection by edge, selection by vertices, star join, and complex join).

3.2.3 System Architecture

The first prototype of our VB-Partitioner framework is implemented on top of a Hadoop cluster. We use Hadoop MapReduce and HDFS to partition heterogeneous graphs and

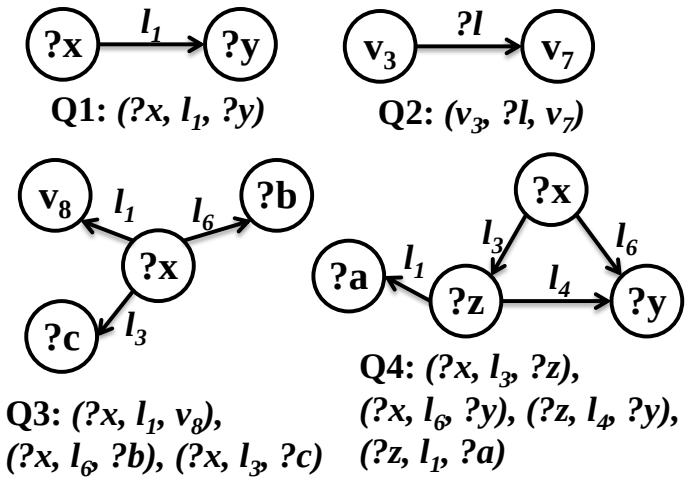


Figure 11: Graph Pattern Query Graphs

manage distributed query execution across a cluster of Hadoop nodes in the Cloud. Fig. 12 shows a sketch of the system architecture. Our system consists of one *master* node and a set of *slave* nodes. When we execute a graph partitioning or distributed query processing algorithm using Hadoop, the master node serves as the NameNode of HDFS and the JobTracker of Hadoop MapReduce. Similarly, the slave nodes serve as the DataNodes of HDFS and the TaskTrackers of Hadoop MapReduce.

Graph Partitioner. Many real-world big graphs exceed the performance capacity (e.g., memory, CPU) of a single node. Thus, we provide a distributed implementation of our VB-Partitioner on a Hadoop cluster of compute nodes. Concretely, we first load the big input graph into HDFS and thus the input graph is split into large HDFS chunks and stored in a cluster of slave nodes. **Extended vertex block generator** generates vertex block or extended vertex block for each vertex in the input graph stored in HDFS using Hadoop MapReduce. **Extended vertex block allocator** performs two tasks to place each vertex block to a slave node of the Hadoop cluster: (i) It employs a vertex block grouping algorithm to assign each extended vertex block to a partition; (ii) It assigns each partition to a slave node, for example using a standard hash function, which will balance the load by attempting to assign equal number of partitions to each slave node. On each slave node, a local graph processing engine is installed to process graph pattern queries against the partitions locally stored on the node. We provide more detail on our graph partitioning

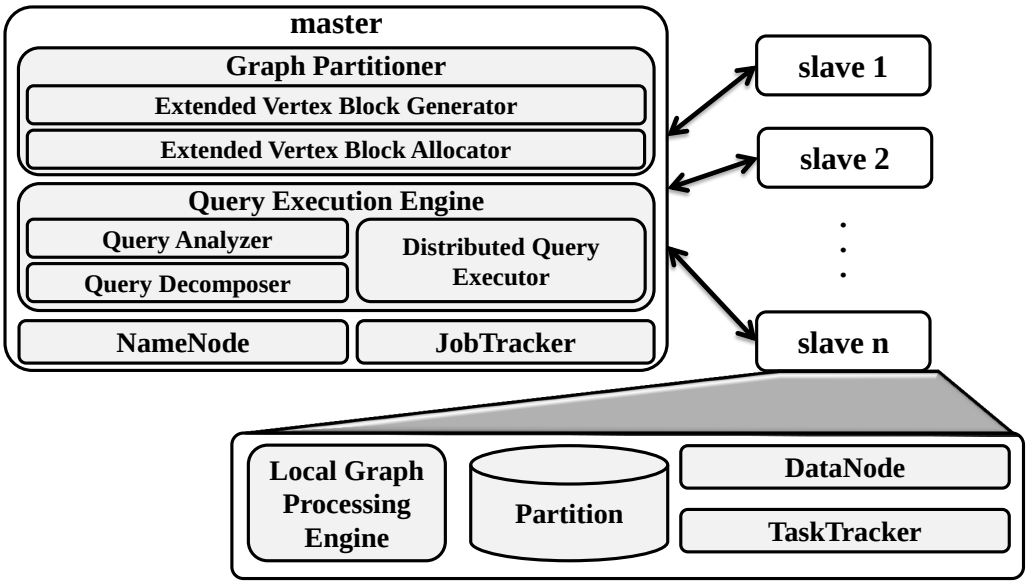


Figure 12: VB-PARTITIONER System Architecture

algorithms in Section 3.3.

Query Execution Engine. To speed up the processing of graph pattern queries, we first categorize our distributed query execution into two types: *intra*-partition processing and *inter*-partition processing. By *intra*-partition processing, we mean that a graph query Q can be fully executed in parallel on each slave node without any cross-node coordination. The only communication cost required to process Q is for the master node to dispatch Q to each slave node. If no global sort of results is required, each slave node can directly (or via its master to) return its locally generated results. Otherwise, either the master node or an elected slave node will be served as the integrator node to merge the partial results received from all slave nodes to generate the final sorted results of Q. By *inter*-partition processing, we mean that a graph query Q as a whole cannot be executed on any slave node, and thus it needs to be decomposed into a set of subqueries such that each subquery can be evaluated by intra-partition processing. Thus, the processing of Q requires multiple rounds of coordination and data transfer across a set of slave nodes. In contrast to intra-partition processing, the network I/O (communication) cost can be extremely high, especially when the number of subqueries is not small and the size of intermediate results to be transferred across the cluster of slave nodes is large.

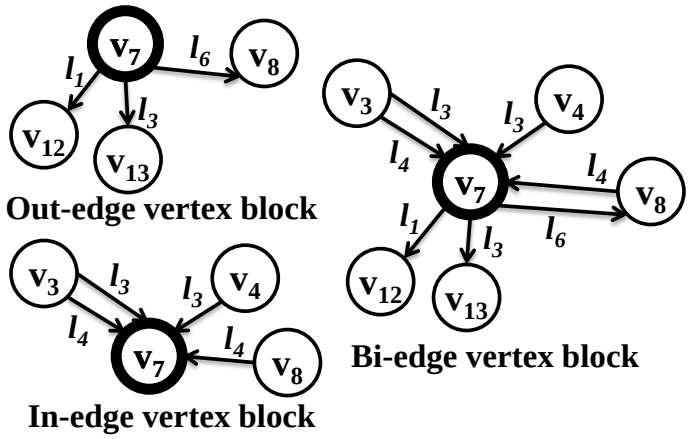


Figure 13: Different Vertex Blocks of v_7

For a given graph query Q, **query analyzer** analyzes Q to see whether Q can be executed using intra-partition processing. If Q can be executed using intra-partition processing, Q is directly sent to distributed query executor. Otherwise, **query decomposer** is invoked to split Q into a set of subqueries such that each subquery can be executed using intra-partition processing. **Distributed query executor** is in charge of executing Q using intra-partition or inter-partition processing by coordinating slave nodes. We will explain our distributed query processing in detail in Section 3.4.

3.3 VB-Partitioner *Framework Design*

The VB-Partitioner framework for heterogeneous graphs consists of three phases. First, we build a vertex block for each vertex in the graph. We guarantee that all the information (vertices and edges) included in a vertex block will be stored in the same partition and thus on the same slave node. Second, we expand each vertex block (VB) to an extended vertex block (EVB). Third, we employ a VB grouping algorithm to assign each VB or EVB to a vertex block-based partition. We below describe each of the three phases in detail.

3.3.1 Vertex Blocks

A vertex block consists of an anchor vertex and its connected edges and vertices. To support customizable and effective data partitioning, we introduce three different vertex blocks based on the direction of connected edges of the anchor vertex: 1) out-edge vertex

block, 2) in-edge vertex block, and 3) bi-edge vertex block. Fig. 13 shows out-edge, in-edge, and bi-edge vertex blocks of vertex v_7 in Fig. 10. We formally define the vertex block as follows.

Definition 14 (Vertex block) Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, **out-edge vertex block** of an anchor vertex $v \in V$ is a subgraph of G which consists of v and all its out-edges, denoted by $VB_v^+ = (V_v^+, E_v^+, \Sigma_{V_v^+}, \Sigma_{E_v^+}, l_{V_v^+}, l_{E_v^+})$ such that $V_v^+ = \{v\} \cup \{v^+ | v^+ \in V, (v, l, v^+) \in E_v^+\}$. Similarly, **in-edge vertex block** of v is defined as $VB_v^- = (V_v^-, E_v^-, \Sigma_{V_v^-}, \Sigma_{E_v^-}, l_{V_v^-}, l_{E_v^-})$ such that $V_v^- = \{v\} \cup \{v^- | v^- \in V, (v^-, l, v) \in E_v^-\}$. Also, **bi-edge vertex block** of v is defined as $VB_v^\pm = (V_v^\pm, E_v^\pm, \Sigma_{V_v^\pm}, \Sigma_{E_v^\pm}, l_{V_v^\pm}, l_{E_v^\pm})$ such that $V_v^\pm = \{v\} \cup \{v^\pm | v^\pm \in V, (v, l, v^\pm) \in E_v^+$ or $(v^\pm, l, v) \in E_v^-\}$.

Each vertex block preserves the basic graph structure of a vertex and thus can be used as an atomic unit (building block) for graph partitioning. By placing a vertex block into the same partition, we can efficiently process all *basic* graph pattern queries using intra-partition processing, such as selection by edge or by vertex, because it guarantees that all vertices and edges required to evaluate such queries are located in the same partition. Consider the graph pattern query $Q2$ $(v_3, ?l, v_7)$ in Fig. 11. We can process the query using intra-partition processing regardless of the type of the vertex block. If we use *out-edge* (or *in-edge*) vertex blocks for partitioning, it is guaranteed that all out-edges (or in-edges) of v_3 (or v_7) are located in the same partition. It is obviously true for *bi-edge* vertex blocks because it is the union of *in-edge* and *out-edge* vertex blocks.

It is worth noting that each partitioning scheme based on each of the three types of vertex blocks can be advantageous for some queries but fail to produce the results of queries effectively. Consider $Q3$ $\{(?x, l_1, v_8), (?x, l_6, ?a), (?x, l_3, ?b)\}$ in Fig. 11. It is guaranteed that all out-edges of any vertex matching `?x` are located in the same partition if we use *out-edge* vertex blocks. This enables the query evaluation using intra-partition processing because only out-edges of `?x` are required. However, if we use *in-edge* vertex blocks, we can no longer evaluate $Q3$ solely using intra-partition processing because we can no longer guarantee that all out-edges of any vertex matching `?x` are located in the same partition.

Consider $Q4$ $\{(?x, l_3, ?z),\ (?x, l_6, ?y),\ (?z, l_4, ?y),\ (?z, l_1, ?a)\}$ in Fig. 11. We cannot process $Q4$ using intra-partition processing because there is no vertex (or vertex variable) that can cover all edges in the query graph using its out-edges, in-edges or even bi-edges. For example, if we consider *bi-edge* vertex block of `?z`, it is clear that there is one remaining edge $((?x, l_6, ?y))$, which cannot be covered by the vertex block. This motivates us to introduce the concept of extended vertex block.

3.3.2 Extended Vertex Blocks

The basic idea of the extended vertex block is to include not only directly connected edges of the anchor vertex but also those within k-hop distance from the anchor vertex. Concretely, to construct the extended vertex block of an anchor vertex, we extend its vertex block hop by hop to include those edges (and their vertices) that are reachable within k hops from the anchor vertex. For example, from the out-edge vertex block of v_7 in Fig. 13, its 2-hop (k=2) extended vertex block will add the out-edges of v_8, v_{12} and v_{13}.

One of the most significant advantages of k-hop extended vertex blocks is that most graph pattern queries can be executed using intra-partition processing without any coordination with another partition. However, when k is too large relative to the size of the graph, extended vertex blocks can be costly in terms of the storage cost on each node. In other words, even though we remove inter-partition communication cost, the slow local processing on each large partition may become the dominating factor for the query processing.

To tackle this problem, we introduce a *k-hop* extended vertex block in which the extension level is controlled by the system parameter k. As a base case, the *1-hop* extended vertex block of an anchor vertex is the same as its vertex block. The *k-hop* extended vertex block of an anchor vertex includes all vertices and edges in its *(k-1)-hop* extended vertex block and additional edges (and their vertices) that are connected to any vertex in the *(k-1)-hop* extended vertex block.

We also define three different types of the k-hop extended vertex block based on the direction of expanded edges: 1) k-hop extended out-edge vertex block, 2) k-hop extended in-edge vertex block, and 3) k-hop extended bi-edge vertex block. Fig. 14 shows the different

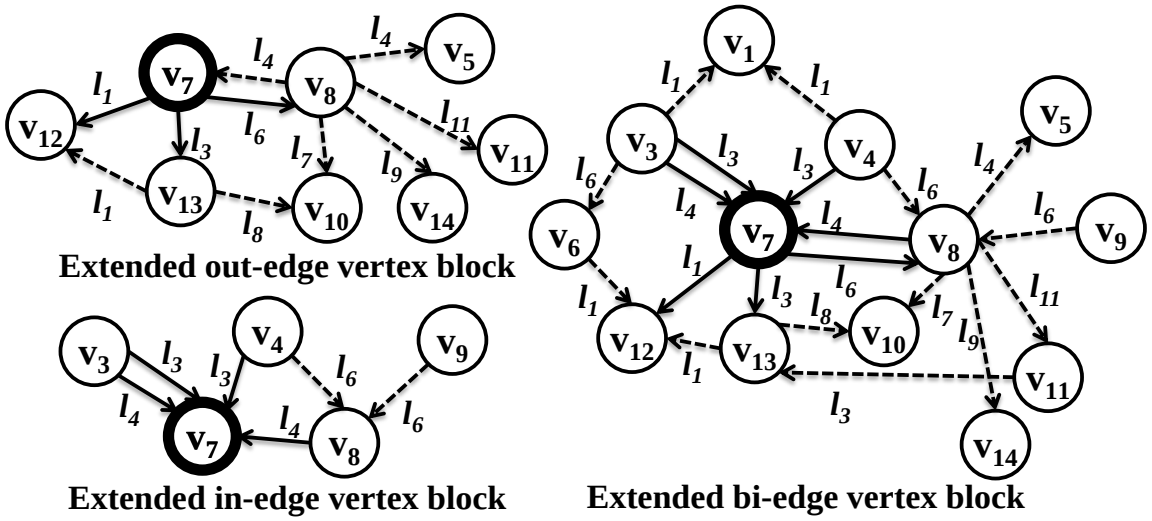


Figure 14: 2-hop Extended Vertex Blocks of v_7

types of 2-hop extended vertex block for v_7. Dotted edges indicate the newly added edges from the corresponding vertex block.

3.3.3 VB-based Grouping Techniques

After we obtain a vertex block or an extended vertex block of each vertex in the input graph, we enter the second phase of VB-Partitioner. It strategically groups a subset of VBs and EVBs into a VB-partition by employing our vertex block-based grouping algorithms such that highly correlated VBs and EVBs will be placed into one VB-partition. We remove any duplicate vertices and edges within each VB-partition.

When assigning each VB or EVB to a partition, we need to consider the following three factors for generating efficient and effective partitions: (i) The generated partitions should be well balanced; (ii) The amount of replications should be small; and (iii) The formation of VB-partitions should be fast and scalable. First, balanced partitions are important for efficient query processing because one big partition, in the imbalanced partitions, can be a bottleneck and increase the overall query processing cost. Second, we need to reduce the number of replicated vertices and edges to construct smaller partitions and thus support faster local query processing in each partition. Since an edge (and its vertices) can be included in several extended vertex blocks, we need to assign those extended vertex blocks sharing many edges to the same partition to reduce the number of replicated edges and

vertices. Third but not the least, we need to support fast partitioning for frequently updated graphs. Since one partitioning technique cannot fit all, we propose three different grouping techniques in which each has its own strength and thus can be accordingly selected for different graphs and query types.

Hashing-based VB Grouping. The hashing-based grouping technique assigns each extended vertex block based on the hash value of the block's anchor vertex name. This partitioning technique generates well-balanced partitions and is very fast. However, the hashing-based VB grouping is not effective in terms of managing and reducing the amount of vertex and edge replication because the hashing-based algorithm pays no attention on the correlation among different VBs and EVBs. If we can develop a smart hash function that is capable of incorporating some domain knowledge about vertex names, we can reduce the number of replicated edges. For example, if we know that vertices sharing the same prefix (or suffix) in their name are closely connected in the input graph, we can develop a new hash function, which uses only the prefix (or suffix) of the vertex names to calculate the hash values, and assign the vertices sharing the common prefix (or suffix) to the same partition.

Minimum cut-based VB Grouping. The minimum cut-based grouping technique utilizes the minimum cut graph partitioning algorithm, which splits an input graph into smaller components by minimizing the number of edges running between the components. After we run the graph partitioning algorithm for an input graph by setting the number of components as the number of partitions, we can get a list that has the assigned component id for each vertex. Since the algorithm assigns each vertex to one component and there is an one-to-one mapping between components and partitions, we can directly utilize the list of components by assigning each VB or EVB to the partition corresponding to the assigned component of its anchor vertex. This grouping technique is very good for reducing the number of replicated edges because we can view the minimum cut algorithm as grouping closely located (or connected) vertices in the same component. Also, because another property of the minimum cut algorithm is to generate uniform components such that the components are of about the same size, this grouping technique can also achieve a good

level of balanced partitions. However, the uniform graph partitioning problem is known to be NP-complete [25]. It often requires a long running time for VB-grouping due to its high time complexity. Our experiments on large graphs in Section 3.5 show that the minimum cut-based VB grouping is practically infeasible for large and complex graphs.

High degree vertex-based VB Grouping. This grouping approach is motivated for providing a better balance between reducing replication and fast processing. The basic idea of this grouping algorithm is to find some high degree vertices with many in-edges and/or out-edges and place the VBs or EVBs of those nearby vertices of each high degree vertex in the same partition of the high degree vertex. By focusing on only high degree vertices, we can effectively reduce the time complexity of grouping algorithm and better control the degree of replications.

Concretely, we first find some high degree vertices whose number of connected edges is larger than a system-supplied threshold value δ. If we increase the δ value, a smaller number of vertices would be selected as the high degree vertices.

Second, for each high degree vertex, we find a set of vertices, called *dependent* vertices, which are connected to the high degree vertex by one hop. There are three types of dependent vertices for each high degree vertex (out-edge, in-edge or bi-edge). If the high degree vertex has an *out-edge* EVB, then we find its dependent vertices by following the *in-edges* of the high degree vertex. Similarly, we check the *out-edges* and *bi-edges* of the high degree vertex for extended *in-edge* and *bi-edge* vertex blocks respectively.

Third, we group each high degree vertex and its dependent vertices to assign them (and their extended vertex blocks) to the same partition. If a vertex is a dependent vertex of multiple high degree vertices, we merge all its high degree vertices and their dependent vertices in the same group. By doing so, we can prevent the replication of the high degree vertices under 2-hop extended *out-edge* vertex blocks. If 3-hop extended *out-edge* vertex blocks are generated, we also extend the dependent vertex set of a high degree vertex by including additional vertices that are connected to any dependent vertex by one hop. We can repeatedly extend the dependent vertex set for $k > 3$. To prevent from generating a huge partition, we exclude those groups, whose size (the number of vertices in the group)

is larger than a threshold value, when we merge groups. By default, we divide the number of all vertices in the input graph by the number of partitions and use the result as the threshold value to identify such huge partitions.

Finally, we assign the extended vertex blocks of all vertices in a high-degree group to the same partition. For each uncovered vertex that is not close to any high degree vertex, we simply select a partition having the smallest size and assign its extended vertex block to that partition.

3.4 Distributed Query Processing

For a given graph pattern query Q, the first step is to analyze Q to determine whether Q can be executed using intra-partition processing or not. If yes, Q is directly sent to the query execution step without invoking the query decomposition step. Otherwise, we iteratively decompose Q into a set of subqueries such that each subquery can be executed using intra-partition processing. Finally, we generate execution plans for Q (intra-partition processing) or for its subqueries (inter-partition processing) and the query result by executing the plans using the cluster of compute nodes.

3.4.1 Query Analysis

In query analysis step, we need to determine whether a query Q needs to be sent to the query decomposer or not. The decision is primarily based on eccentricity, radius, and center vertex in the context of graph.

Definition 15 (Eccentricity) Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, the out-edge **eccentricity** ϵ^+ of a vertex $v \in V$ is the greatest out-edge hop count from v to any edge in G and formally defined as follows:

$$\epsilon^+(v) = \max_{(w,l,w') \in E} hop^+(v, wlw')$$

The in-edge eccentricity ϵ^- and bi-edge eccentricity $\epsilon^\pm$ are similarly defined. The eccentricity of a vertex in a graph can be thought of as how far a vertex is from the vertex most distant from it in the graph.

Definition 16 (Radius and Center vertex) Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, the out-edge **radius** of G, denoted by $r^+(G)$, is the minimum out-edge eccentricity of any vertex $v \in V$ and formally defined as follows:

$$r^+(G) = \min_{v \in V} \epsilon^+(v)$$

The out-edge **center vertices** of G, denoted by $CV^+(G)$, are the vertices whose out-edge eccentricity equals to the out-edge radius of G and formally defined as follows:

$$CV^+(G) = \{v | v \in V, \epsilon^+(v) = r^+(G)\}$$

The in-edge radius $r^-(G)$, in-edge center vertices $CV^-(G)$, bi-edge radius $r^\pm(G)$, and bi-edge center vertices $CV^\pm(G)$ are similarly defined.

Assuming that the partitions are constructed using k-hop extended vertex blocks, for a graph pattern query Q and its query graph G_Q, we first calculate the *radius* and the center vertices of the query graph based on Definition 16. If the partitions are constructed using extended out-edge (in-edge or bi-edge) vertex blocks, we calculate $r^+(G_Q)$ ($r^-(G_Q)$ or $r^\pm(G_Q)$) and $CV^+(G_Q)$ ($CV^-(G_Q)$ or $CV^\pm(G_Q)$). If the radius is equal to or less than k, then the query Q as a whole can be executed using the intra-partition processing. This is because, from the center vertices of G_Q, our k-hop extended vertex blocks guarantee that all edges that are required to evaluate Q are located in the same partition. In other words, by choosing one of the center vertices as an anchor vertex, it is guaranteed that the k-hop extended vertex block of the anchor vertex covers all the edges in G_Q given that the radius of G_Q is not larger than k. Therefore we can execute Q without any coordination and data transfer among the partitions. If the radius is larger than k, we need to decompose Q into a set of subqueries.

Fig. 15 presents how our query analysis step works under three different types (out-edge, in-edge and bi-edge) of extended vertex blocks for graph pattern query $Q4$ in Fig. 11. The eccentricity value of each vertex is given next to the vertex. Since the **out-edge** radius of the query graph is 2, we can execute the query using intra-partition processing if the partitions are constructed using k-hop extended *out-edge* vertex blocks and k is equal to or

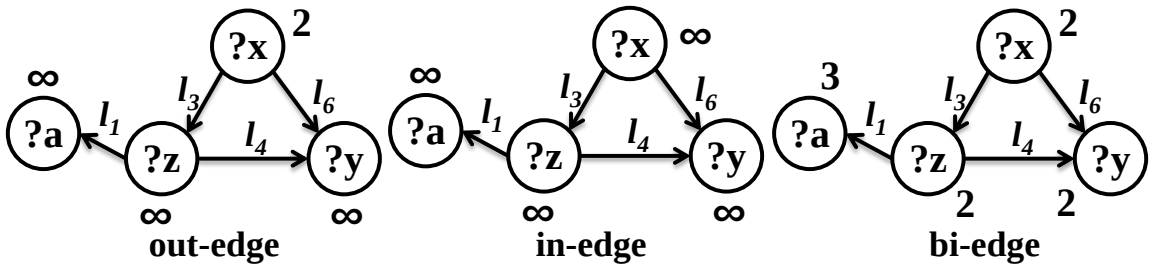


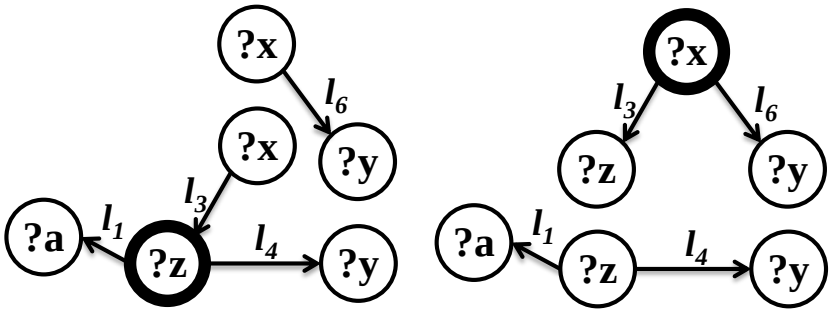
Figure 15: Query Analysis

larger than 2. However, the **in-edge** radius of the query graph is *infinity* because there is no vertex that has at least one in-edge path to all the other vertices. Therefore, we cannot execute the query using intra-partition processing if the partitions are constructed using extended *in-edge* vertex blocks.

3.4.2 Query Decomposition

To execute a graph pattern query Q using *inter-partition* processing, it is necessary to slit Q into a set of subqueries in which each subquery can be executed using intra-partition processing. Given that we use Hadoop and HDFS to join the partial results generated from the subqueries, we need to carefully decompose Q in order to minimize the join processing cost. Since we use one Hadoop job to join two sets of partial results and each Hadoop job has an initialization overhead of about 10 seconds regardless of the input data size, we decompose Q by minimizing the number of subqueries. To find such decomposition, we use an intuitive approach that first checks whether Q can be decomposed into *two* subqueries such that each subquery can be evaluated using intra-partition processing. To check whether a subquery can be executed using intra-partition processing, we calculate the radius of the subquery's graph and then perform the query analysis steps outlined in the previous section. We repeat this process until at least one satisfying decomposition is found.

Concretely, we start the query decomposition by putting all vertices in the query graph G_Q of Q into a set of candidate vertices to be examined in order to find such a decomposition having two subqueries. For each candidate vertex v, we draw the k-hop extended vertex block of v in G_Q, assuming that the partitions are constructed using k-hop extended vertex blocks. For the remaining edges of G_Q, which are not covered by the k-hop extended vertex



(a) from `?z` (b) from `?x`

Figure 16: Query Decomposition (bi-edge)

block of v, we check whether there is any other candidate vertex whose k-hop extended vertex block in G_Q can fully cover the remaining edges. If there is such a decomposition, we treat each subgraph as a subquery of Q. Otherwise, we increase the number of subqueries by one and then repeat the above process until we find a satisfying decomposition. If we find more than one satisfying decompositions having the equal number of subqueries, then we choose the one in which the standard deviation of the size (i.e., the number of edges) of subqueries is the smallest, under the assumption that a small subquery may generate large intermediate results. We leave as future work the query optimization problem where we can utilize additional metadata such as query selectivity information.

For example, let us assume that the partitions are constructed using 1-hop extended *bi-edge* vertex blocks and thus graph pattern query $Q4$ in Fig. 11 cannot be executed using intra-partition processing. To decompose the query, if we start with vertex `?z`, we will get a decomposition that consists of two subqueries as shown in Fig. 16(a). If we start with vertex `?x`, we will also get two subqueries as shown in Fig. 16(b). Based on the smallest subquery standard deviation criterion outlined above, we choose the latter because two subqueries are of the same size.

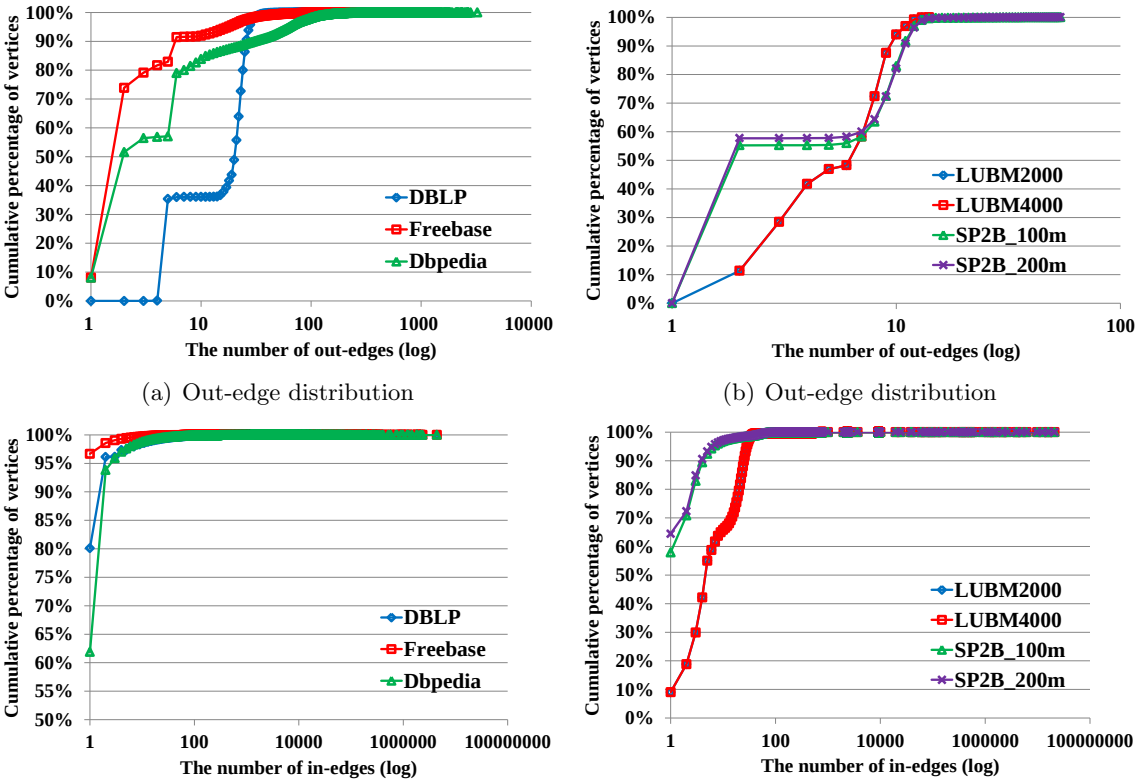
3.5 Experimental Evaluation

In this section, we report the experimental evaluation results of our partitioning framework for various heterogeneous graphs. We first explain the characteristics of datasets we used for our evaluation and the experimental settings. We divide the experimental results into four

categories: (i) We show the **data partitioning and loading time** for different extended vertex blocks and grouping techniques and compare it with the data loading time in a single server. (ii) We present the **balance and replication level** of generated partitions using different extended vertex blocks and grouping techniques. (iii) We conduct the experiments on **query processing latency** using various types of graph pattern queries. (iv) We also evaluate the scalability of our partitioning framework by increasing the dataset size and the number of servers in the cluster.

3.5.1 Datasets

To show the working of our partitioning framework for various graphs having totally different characteristics, we not only use three real graphs but also generate three graphs from each of two different benchmark generators. As real graphs, we choose **DBLP** [1] containing bibliographic information in computer science, **Freebase** [5] including a large knowledge base, and **DBpedia** [7] having structured information from Wikipedia. As benchmark graphs, we choose LUBM and SP^{2}Bench, which are widely used for evaluating RDF storage systems, and generate **LUBM2000**, **LUBM4000**, **LUBM8000** using LUBM and **SP2B-100M**, **SP2B-200M** and **SP2B-500M** using SP^{2}Bench. As a data cleaning step, we remove any duplicate edges using one Hadoop job for each dataset. Table 8 shows the number of vertices and edges and the average number of out-edges and in-edges of the datasets. Note that the benchmark datasets, generated from the same benchmark generator, have almost the same average number of out-edges and in-edges regardless of the dataset size. Fig. 17 shows the out-edge and in-edge distribution of the datasets. In the x-axis of the figures, we plot the number of out-edges (or in-edges) and in the y-axis we plot the percentage of vertices whose number of out-edges (or in-edges) is equal to or less than this number of out-edges (or in-edges). For example, about 85%, 97%, and 89% of vertices have 25 or less out-edges on DBLP, Freebase, and DBpedia respectively. Note that the benchmark datasets, generated from the same benchmark generator, have almost the same out-edge and in-edge distribution regardless of the dataset size. We omit the results of LUBM8000 and SP2B-500M because each has almost the same distribution with datasets



(a) Out-edge distribution

(b) Out-edge distribution

(c) In-edge distribution

(d) In-edge distribution

Figure 17: Out-edge and In-edge Distribution

from the same benchmark generator.

Table 8: Datasets (VB-PARTITIONER)

Dataset	#vertices	#edges	avg. #out	avg. #in
DBLP	25,901,515	56,704,672	16.66	2.39
Freebase	51,295,293	100,692,511	4.41	2.11
DBpedia	104,351,705	287,957,640	11.62	2.82
LUBM2000	65,724,613	266,947,598	6.15	8.27
LUBM4000	131,484,665	534,043,573	6.15	8.27
LUBM8000	262,973,129	1,068,074,675	6.15	8.27
SP2B-100M	55,182,878	100,000,380	5.61	2.11
SP2B-200M	111,027,855	200,000,007	5.49	2.08
SP2B-500M	280,908,393	500,000,912	5.31	2.04

3.5.2 Setup

We use a cluster of 21 nodes (one is the master node) on Emulab [115]: each has 12 GB RAM, one 2.4 GHz 64-bit quad core Xeon E5530 processor, and two 7200 rpm SATA disks (250GB and 500GB). The network bandwidth is about 40 MB/s. When we measure the query processing time, we perform five cold runs under the same setting and show the *fastest*

time to remove any possible bias posed by OS and/or network activity.

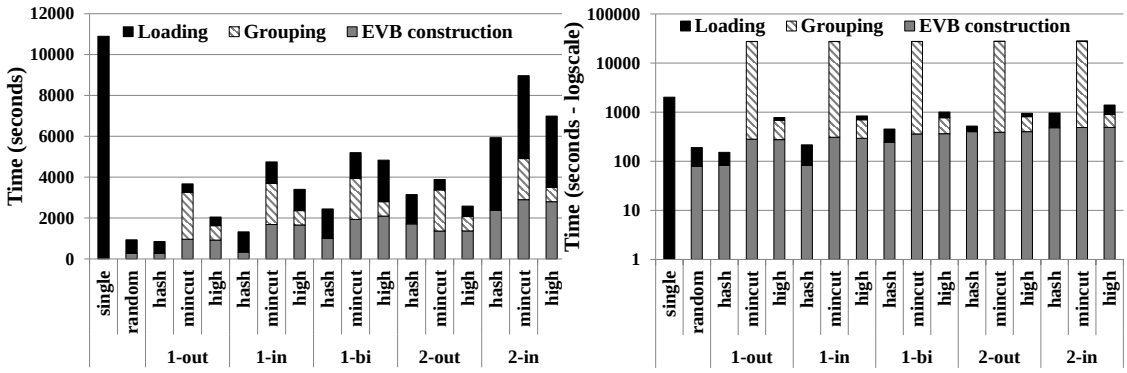
As a local graph processing engine, we install RDF-3X version 0.3.5 [86], on each slave server, which is an open-source RDF management system. We use Hadoop version 1.0.4 running on Java 1.6.0 to run our graph partitioning algorithms and join the intermediate results, generated by subqueries, during inter-partition processing. For comparison, we also implement random partitioning. To implement the minimum-cut based VB grouping technique, we use graph partitioner METIS version 5.0.2 [15] with its default configuration.

To simplify the name of our extended vertex blocks and grouping techniques, we use `[k]-[out|in|bi]-[hash|mincut|high]` as our naming convention. For example, *1-out-high* indicates the high degree vertex-based technique with 1-hop extended out-edge vertex blocks.

3.5.3 Partitioning and Loading Time

We first compare the partitioning and loading time of our framework with that on a single server. Fig. 18 shows the partitioning and loading time of LUBM2000 and DBLP for different extended vertex blocks and grouping techniques. The loading time indicates the loading time of RDF-3X. The single server approach has only the loading time because there is no partitioning. To support efficient partitioning, we implement the extended vertex block construction and grouping using Hadoop MapReduce in the cluster of nodes. Since the hashing-based grouping technique simply uses a hash function (By default, we use the hash function of Java String class) to assign each extended vertex block to a partition, we incorporate the grouping step into the construction step and thus there is no grouping time for those using the hashing-based grouping technique. The grouping time of the minimum cut-based grouping technique includes both the input conversion time (from RDF to METIS input format) and METIS running time. We also implement the input conversion step using Hadoop MapReduce in the cluster of nodes for efficient conversion.

Fig. 18(a) clearly shows that we can significantly reduce the graph loading time by using our partitioning framework, compared to using only single server. The only exception is when we use the minimum cut-based grouping technique in which we need to convert



(a) LUBM2000 (b) DBLP

Figure 18: Partitioning and Loading Time

the datasets into the METIS input formats, as shown in Fig. 18(b). The conversion time depends on not only the dataset size but also the structure of the graph. For example, the conversion times of DBLP and Freebase are about 7.5 hours and 35 hours respectively, which are much longer than 50 minutes of DBpedia even though DBpedia has much more edges. We think that this is because DBLP and Freebase include some vertices having a huge number of connected edges. For example, there are 4 and 6 vertices having more than one million in-edges on DBLP and Freebase respectively. Also note that the minimum cut-based grouping technique could not work on LUBM4000, LUBM8000, and SP2B-500M because METIS failed due to the insufficient memory on a single machine with 12 GB RAM. This result indicates that the minimum cut-based grouping technique is infeasible for some graphs having a huge number of vertices and edges and/or complex structure.

3.5.4 Balance and Replication level

To show the balance of generated partitions in terms of the number of edges, we use the *relative standard deviation expressed as a percentage*, defined as the ratio of the standard deviation to the mean (and then multiplied by 100 to be expressed as a percentage). A higher percentage means that the generated partitions are less balanced. Fig. 19 shows the relative standard deviation for different extended vertex blocks and grouping techniques. As we expect, the hashing-based grouping technique generates the most balanced partitions for most cases. Especially, using extended *out-edge* vertex blocks, the hashing-based technique

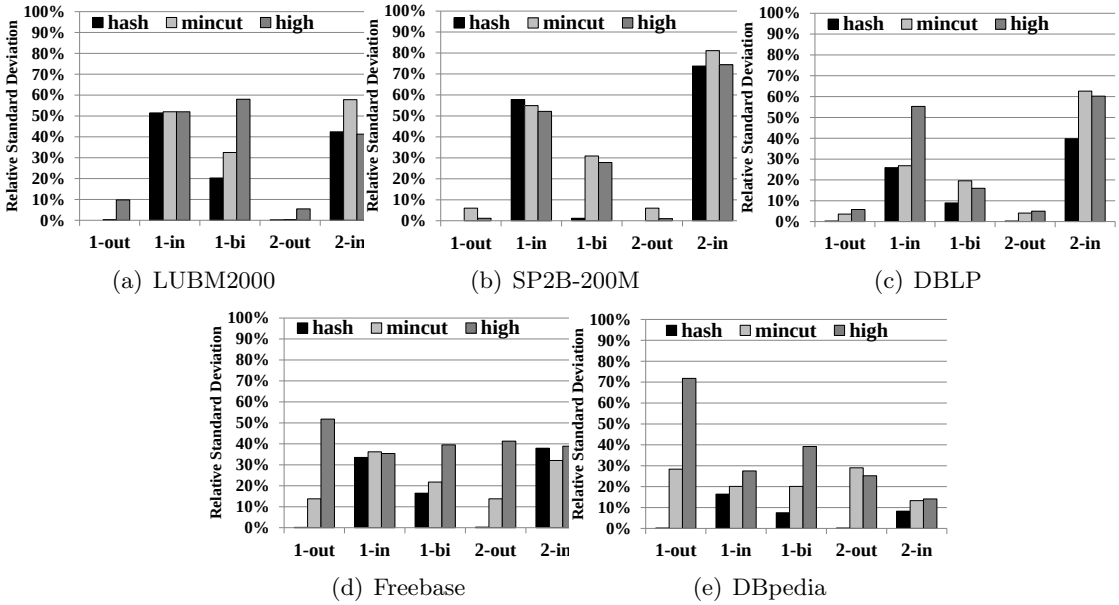


Figure 19: Balance of Generated Partitions

constructs almost perfectly balanced partitions. It is interesting to note that, the hashing-based technique using extended in-edge vertex blocks generates less balanced partitions than that using out-edge EVBs. This is because there are some vertices having a huge number of in-edges (e.g., more than one million in-edges) as shown in Fig. 17(c) and Fig. 17(d). Therefore, partitions including the extended vertex blocks of such vertices will have much more edges than the others. We omit the results of LUBM4000, LUBM8000, SP2B-500M, and SP2B-500M because each has almost the same relative standard deviation with the dataset from the same benchmark generator.

To see how many edges are replicated, Fig. 20 shows the total number of edges of all the generated partitions for different extended vertex blocks and grouping techniques. As we expect, the minimum cut-based grouping technique is the best in terms of reducing the replication. Especially, when we use 2-hop out-edge EVBs, the minimum cut-based grouping technique replicates only a small number of edges. However, for the other vertex blocks, the benefit of the minimum cut-based grouping technique is not so significant if we consider its overhead as shown in Fig. 18. Also recall that the minimum cut-based grouping technique fails to work on LUBM4000, LUBM8000, and SP2B-500M because METIS failed due to the insufficient memory.

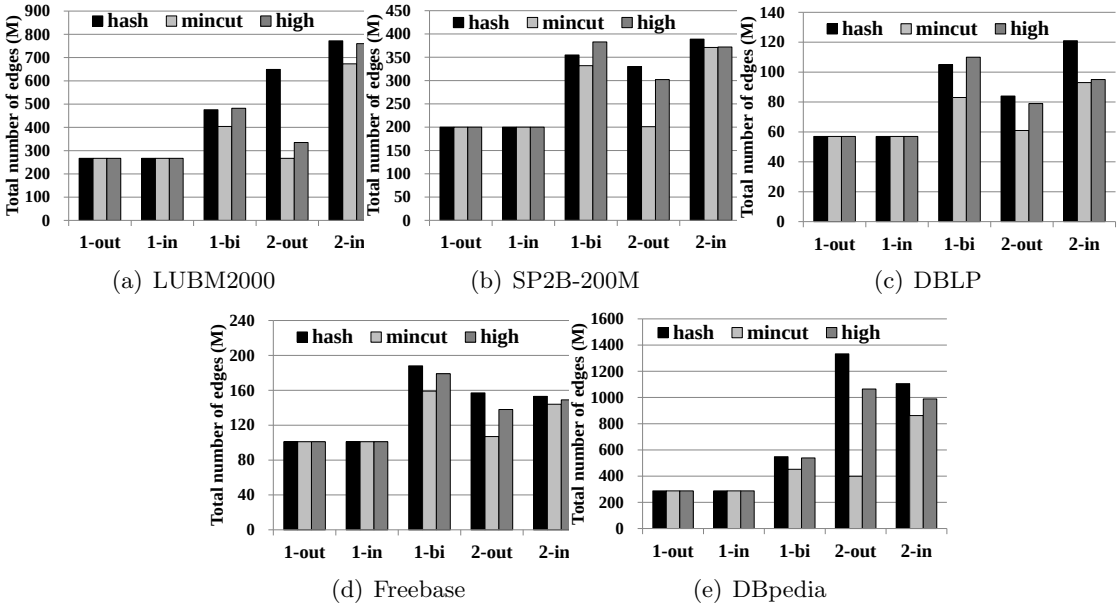


Figure 20: Replication Level

3.5.5 Query Processing

Since LUBM provides 14 benchmark queries, we utilize them to evaluate query processing in the partitions generated by our partitioning framework. Among 14 queries, two queries (Q6 and Q14) are basic graph pattern queries (i.e., only one edge in their query graph) and 6 queries (Q1, Q3, Q4, Q5, Q10, and Q13) are star-like queries in which all the edges in their query graph are out-edges from one vertex variable. Fig. 21 shows the graph pattern query graphs for the other queries. We omit the edge label because there is no edge variable.

Fig. 22 shows the query processing time of all 14 benchmark queries for different extended vertex blocks and grouping techniques on LUBM2000. For brevity, we omit the results of using 1-hop and 2-hop extended *in-edge* vertex blocks because they are not adequate for the benchmark queries due to many leaf-like vertices that have only one in-edge and no out-edge. All partitioning approaches using 1-hop out-edge EVBs, 1-hop bi-edge EVBs, and 2-hop out-edge EVBs ensure intra-partition processing for the basic graph pattern queries (Q6 and Q14) and star-like queries (Q1, Q3, Q4, Q5, Q10, and Q13). Among the remaining queries (Q2, Q7, Q8, Q9, Q11, and Q12), no query can be executed using intra-partition processing over 1-hop extended out-edge and bi-edge vertex blocks. On the

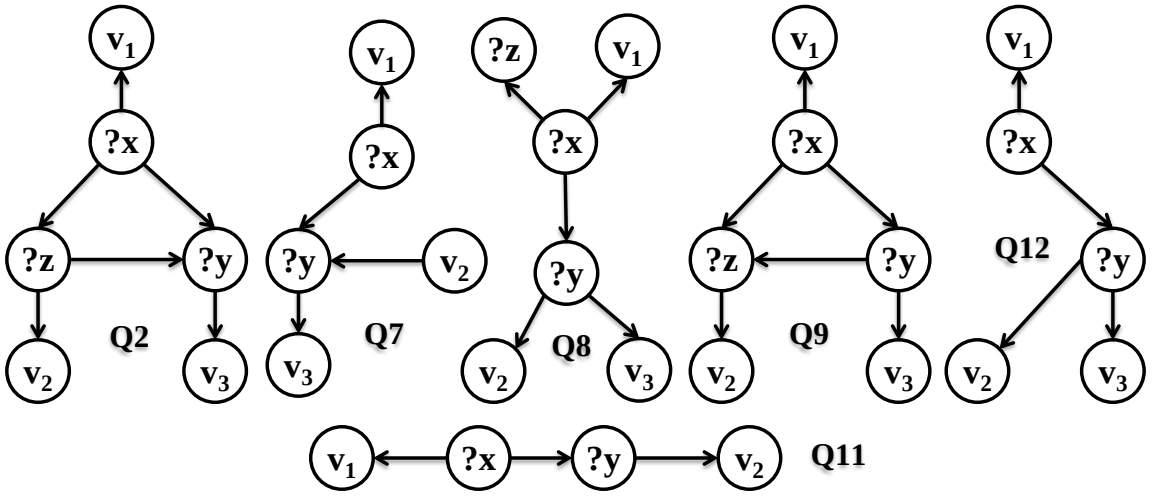
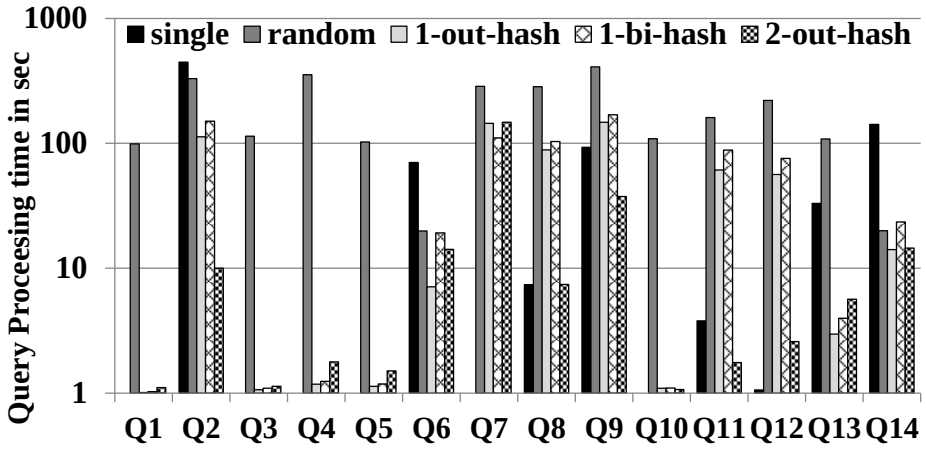


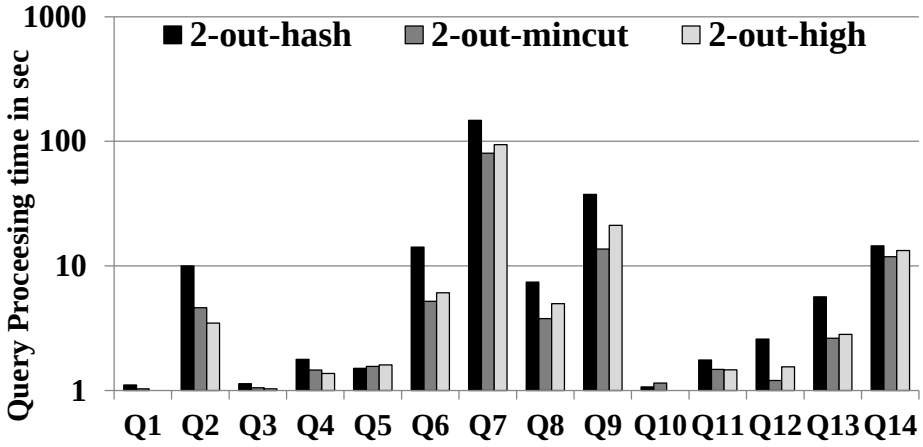
Figure 21: Benchmark Query Graphs

other hand, 2-hop out-edge EVBs guarantee intra-partition processing for all the benchmark queries except Q7, in which 2-hop extended out-edge vertex block of `?x` cannot cover the edge from v_2 to `?y`.

The result clearly shows the huge benefit of intra-partition processing, compared to inter-partition processing. For example, for Q2, the query processing time over 2-hop out-edge EVBs is only 4% of that over 1-hop out-edge EVBs as shown in Fig. 22(a). That is two orders of magnitude faster than the result on a single server. If we use inter-partition processing, it is much slower than using intra-partition processing due to the initialization overhead of Hadoop and the large size of intermediate results. For example, the size of the intermediate results for Q7 over 2-hop out-edge EVBs is 1.2 GB, which is much larger than the final result size of 907 bytes. The result for Q7 also shows the importance of the number of subqueires in inter-partition processing. The query processing over 2-hop out-edge EVBs, which consists of 2 subqueries, is only 65% of that over 1-hop out-edge EVBs, which consists of 3 subqueries, even though the partitions generated using 2-hop out-edge EVBs are much larger as shown in Fig. 20(a). For star queries Q1, Q3, Q4, Q5, and Q10 having very high selectivity (i.e., the result size is less than 10kb), the query processing is usually fast (less than 2 seconds) in the partitions generated by our framework. However, it is slight slower than the query processing on a single server because there is some overhead on the master node, which sends the query to all the slave nodes and merges the partial results received



(a) Effects of different extended vertex blocks

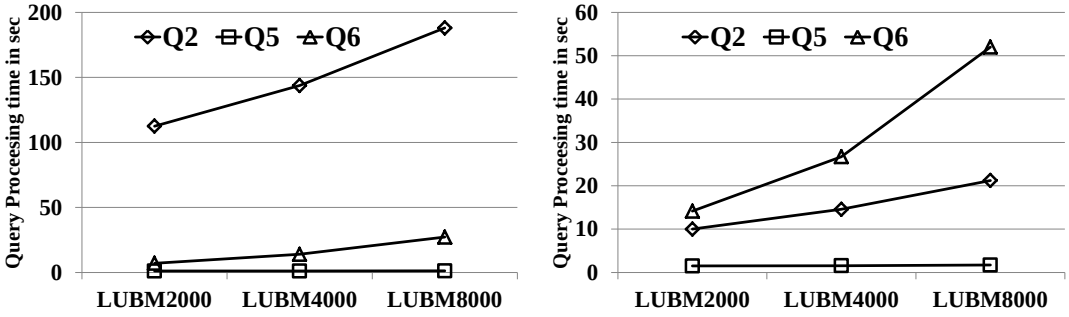


(b) Effects of different grouping techniques

Figure 22: Query Processing Time on LUBM2000

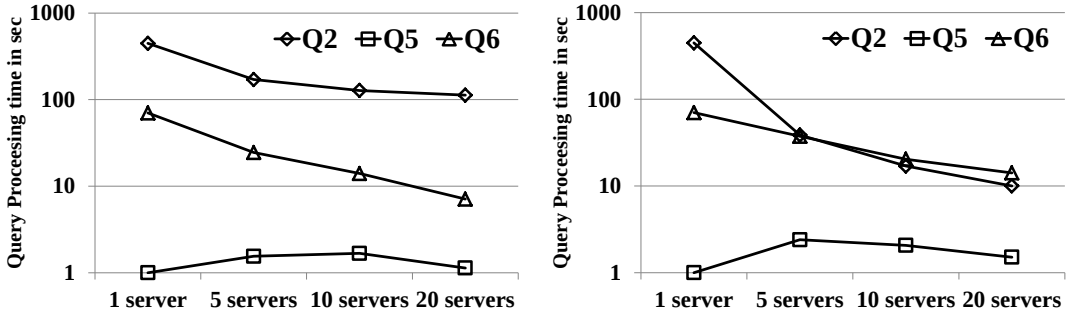
from the slave nodes. When we measure the query processing time on a single server, there is no network cost because queries are requested and executed in the same server.

Fig. 22(b) shows the effect of different grouping techniques using the same extended vertex blocks (i.e., the guarantee of intra-partition processing is the same). The result indicates that the query processing depends on the replication level of the generated partitions. The query processing in the partitions generated using the minimum cut-based grouping technique is usually faster because the minimum cut-based technique generates smaller partitions than the others as shown in Fig. 20.



(a) 1-out-hash (b) 2-out-hash

Figure 23: Scalability with Varying Dataset sizes



(a) 1-out-hash (b) 2-out-hash

Figure 24: Scalability with Varying Cluster sizes

3.5.6 Scalability

To evaluate the scalability of our partitioning framework, we report the query processing results with varying dataset size in Fig. 23. For brevity, we choose one basic graph pattern query (Q6), one star-like query (Q5), and one complex query (Q2). The increase of the query processing time for Q6 is almost proportional to the dataset size and there is only slight increase for Q5 because its results are the same regardless of the dataset size. For Q2, there is only slight increase over 2-hop out-edge EVBs (Fig. 23(b)). However, there is a considerable increase over 1-hop out-edge EVBs because much more intermediate results are generated, compared to the increase of the final results.

Fig. 24 shows the results of another scalability experiment with varying numbers of slave nodes from 5 to 20 on LUBM2000. Note that the results on 1 server represent the query processing time without our partitioning and the query processing times are displayed as log scale. There is almost no big decrease for Q5 because it is already a fast query on 5

servers. We can see the considerable reduction of query processing time for Q2 and Q6 with an increasing number of servers, primarily due to the reduced partition size. However, as shown in the results of Q2 over 1-hop out-edge EVBs (Fig. 24(a)), there would be a point where adding more servers does not improve the query processing time any more because the transfer time of lots of results to the master node is unavoidable.

3.6 Related Work

To process large graphs in a cluster of compute nodes, several graph computation models based on vertex-centric approaches have been proposed in recent years, such as Pregel [80] and GraphLab [77]. Also, GraphChi [67] has been proposed to process large graphs on a single computer in reasonable time. Even though they can efficiently process some famous graph operations, such as PageRank and shortest path computations, they are not adequate for general graph pattern queries (i.e., subgraph matching) in which fast query processing (sometimes a couple of seconds) is preferred by evaluating small parts of input graphs. This is primarily because their approaches are based on multiple iterations and optimized for specific graph operations in which all (or most) vertices in a graph participate in the operations. Our partitioning framework focuses on efficient and effective partitioning for processing general graph pattern queries on large heterogeneous graphs.

Graph partitioning has been extensively studied in several communities for several decades [61, 52, 62, 63]. A typical graph partitioner divides a graph into smaller partitions that have minimum connections between them, as adopted by METIS [61, 63, 15] or Chaco[52, 6]. Various efforts in graph partitioning research have been dedicated to partitioning a graph into similar sized partitions such that the workload of servers storing these partitions will be more or less balanced. We utilize the results of one famous graph partitioner (METIS) to implement one of our grouping techniques to group our extended vertex blocks.

In recent years, a few techniques have been proposed to process RDF graphs in a cluster of compute nodes. [97, 56] directly store RDF triples (edges) in HDFS as flat text files to process RDF queries. [43] utilizes HBase, a column-oriented store modeled after Google's

Bigtable, to store and query RDF graphs. Because general file system-based storage layers, such as HDFS, are not optimized for graph data, their query processing is much less efficient than those using a local graph processing engine, as reported in [55]. Also, because their query processing heavily depends on multiple rounds of inter-machine communication, they usually incur long query latencies. [55] utilizes the results of an existing graph partitioner to partition RDF graphs and stores generated partitions on RDF-3X to process RDF queries locally. As we reported in Section 3.5, running an existing graph partitioner has a large amount of overhead for huge graphs (or graphs having complex structure) and may not even be practically feasible for some large graphs.

3.7 Conclusion

We present VB-Partitioner – a distributed data partitioning model and algorithms for efficient processing of queries over large-scale graphs in the Cloud. This chapter makes three original contributions. First, we introduce the concept of vertex blocks (VBs) and extended vertex blocks (EVBs) as the building blocks for semantic partitioning of large graphs. Second, we describe how VB-Partitioner utilizes vertex block grouping algorithms to place those vertex blocks that have high correlation in graph structure into the same partition. Third, we develop a VB-partition guided query partitioning model to speed up the parallel processing of graph pattern queries by reducing the amount of inter-partition query processing. We evaluate our VB-Partitioner through extensive experiments on several real-world graphs with millions of vertices and billions of edges. Our results show that VB-Partitioner significantly outperforms the popular random block-based data partitioner in terms of query latency and scalability over large-scale graphs.

Our research effort continues along several directions. The first prototype implementation of VB-Partitioner is on top of Hadoop Distributed File System (HDFS) with RDF-3X [86] installed on every node of the Hadoop cluster as the local storage system. We are interested in replacing RDF-3X by TripleBit [119] or GraphChi [67] as the local graph store to compare and understand how different choices of local stores may impact

on the overall performance of our VB-Partitioner. In addition, we are working on efficient mechanisms for deploying and extending our VB-Partitioner to speed up the set of iterative graph algorithms, including shortest paths, PageRank, and random walk-based graph clustering. For example, Pregel [80] can speed up the set of graph computations that are centered on out-edge vertex blocks such as shortest path discovery, and GraphChi [67] can speed up those iterative graph computations that rely on in-edge vertex blocks, such as PageRank and triangle counting. We conjecture that our VB-Partitioner can be effective for a broader range of iterative graph operations. Furthermore, we are also working on extending Hadoop MapReduce programming model and library to enable fast graph operations, ranging from graph queries and reasoning to iterative graph algorithms.

CHAPTER IV

GRAPHMAP: SCALABLE ITERATIVE GRAPH COMPUTATION FRAMEWORK

Scaling large-scale graph processing has been a heated systems research topic in recent years. Existing distributed graph processing systems, such as Pregel, are based solely on distributed memory for their computations and fail to provide seamless scalability when the graph data and their intermediate computation results no longer fit into the memory. Most existing distributed approaches for iterative graph computations to date do not consider utilizing secondary storage as a viable solution. In this chapter we present GraphMap, a distributed iterative graph computation framework, which effectively utilizes secondary storage to maximize access locality and speed up distributed iterative graph computations. GraphMap has three salient features: (1) We distinguish those data states that are mutable during iterative computations from those that are read-only in all iterations to maximize sequential accesses and minimize random accesses. (2) We devise a two-level graph partitioning algorithm to enable balanced workloads and locality-optimized data placement. (3) We propose a suite of locality-based optimizations to improve computation efficiency. Extensive experiments on several real-world graphs show that GraphMap outperforms existing distributed memory-based systems for various iterative graph algorithms.

4.1 Introduction

Graphs are pervasively used for modeling information networks of real-world entities with sophisticated relationships. Many applications from science and engineering to business domains use iterative graph computations to analyze large graphs and derive deep insight from a huge number of implicit relationships among entities. Considerable research effort on scaling large graph computations has been devoted to two different directions. One is to deploy a super powerful many-core computer with memory capacity of hundreds or thousands of gigabytes [91] and another is to explore the feasibility of using a cluster of

distributed commodity servers.

Most of research effort on deploying a supercomputer for fast iterative graph computations assumes considerable computing resources and thus focuses primarily on parallel optimization techniques that can maximize parallelism among many cores and tasks performed on the cores. A big research challenge for efficient many-core computing is the tradeoff between the opportunities for massive parallel computing and the cost of massive synchronization for multiple iterations, which tend to make the overall performance of parallel processing significantly less optimal at the high ownership cost of supercomputers.

As commodity computers become pervasive for many scientists and small or medium enterprise organizations, we witness a rapidly growing demand for distributed iterative graph computations on a cluster of commodity servers. Google Pregel [80] and its open source implementations − Apache Giraph [3] and Hama [4] − have shown remarkable initial success. However, existing distributed graph processing systems, represented by Pregel, heavily rely on distributed memory-based computation model. Concretely, a large graph is first distributed using random or hash partitioning to achieve data-level load balance. Then each compute node of the cluster needs to be able to not only load the entire local graph but also hold both the intermediate results of iterative computations and all the communication messages it needs to send to and receive from every other node of the cluster. Thus, existing approaches suffer from poor scalability when the weakest compute node in the cluster fails to hold the local graph and all the intermediate (computation and communication) results in memory. The dilemma lies in the fact that simply increasing the size of the compute cluster often fails for iterative computations on large graphs. This is because, with a larger cluster, one can reduce the size of the graph that needs to be held in memory at the price of significantly increased amount of distributed messages each node needs to send to and receive from a larger number of nodes in the cluster. For example, a recent study [108] shows that computing 5 iterations of PageRank on twitter 2010 dataset with 42 millions of vertices and 1.5 billions of edges takes 487 seconds on a Spark [121] cluster of 50 nodes and 100 CPUs. Another study [110] shows that counting triangles on twitter 2010 dataset takes 423 minutes on a large Hadoop cluster of 1636 nodes. Surprisingly, most of the existing

approaches for iterative graph computations on a cluster of servers do not explore the option of integrating secondary storage as a viable solution due to a potentially large number of random disk accesses.

In this chapter we argue that secondary storage can play an important role in maximizing both in-memory and on-disk access locality for running iterative graph algorithms on large graphs. By combining it with efficient processing at each local node of a compute cluster, one can perform iterative graph computations more efficiently than the distributed memory-based model in many cases. The ability to intelligently integrate secondary storage into the cluster computing infrastructure for memory intensive iterative graph computations can be beneficial from multiple dimensions: (i) With efficient management of in-memory and on-disk data, one can not only reduce the size of the graph partitions to be held at each node of the cluster but also match the performance of the distributed memory-based system. (ii) One can carry out expensive iterative graph computations on large graphs using a much smaller and affordable cluster (tens of nodes), instead of relying on the availability of a large cluster with hundreds or thousands of compute nodes, which is still costly even with pay-as-you-go elastic cloud computing pricing model.

With these problems and design objectives in mind, we develop GraphMap, a distributed iterative graph computation framework, which can effectively utilize secondary storage for memory-intensive iterative graph computations by maximizing in-memory and on-disk access locality. GrapMap by design has three salient features. First, we distinguish those data states that are mutable during iterative computations from those that are read-only during iterative computations to maximize sequential accesses and minimize random accesses. We show that by keeping mutable data in memory and read-only data on secondary storage, we can significantly improve disk IO performance by minimizing random disk IOs. Second, we support three types of vertex blocks (VBs) for each vertex: in-VB for in-edges of a vertex, out-VB for out-edges of a vertex, and bi-VB for all edges of a vertex and devise a two-level graph partitioning algorithm to enable balanced workloads and locality-optimized data placement. Concretely, we use hash partitioning to distribute vertices and their vertex blocks to different compute nodes and then use range partitioning

to group the vertices and their vertex blocks within each hash partition into storage chunks of fixed size. Last but not the least, we devise a suite of locality-based optimizations to improve computation efficiency, including progressive pruning of non-active vertices and edges to reduce the unnecessary memory and CPU consumption, partition-aware identifier assignment, partition-aware message batching, and local merge of partial updates. These design features enable GraphMap to achieve two objectives at the same time: (1) to minimize non-sequential disk IOs, and significantly improve the secondary storage performance, making the integration of external storage a viable solution for distributed processing of large graphs; and (2) to minimize the communication cost among different graph partitions and maximize the overall computation efficiency of iterative graph algorithms. We evaluate GraphMap on a number of real graph datasets using several graph algorithms by comparing with existing representative distributed memory-based graph processing systems, such as Apache Hama. Our experimental results show that GraphMap outperforms existing distributed graph systems for various iterative graph algorithms.

4.2 GraphMap *Overview*

In this section, we first define some basic concepts used in GraphMap and then provide an overview of GraphMap including its partitioning technique, programming API, and system architecture.

4.2.1 Basic Concepts

GraphMap models all information networks as directed graphs. For an undirected graph, we convert each undirected edge into two directed edges.

Definition 17 (Graph) A graph is denoted by $G = (V, E)$ where V is a set of vertices and E is a set of directed edges (i.e., $E \subseteq V \times V$). For an edge e such that $\{e = (u, v) \in E, u, v \in V\}$, we call u and v the *source vertex* and *destination vertex* of e respectively. e is an *in-edge* of vertex v and an *out-edge* of vertex u. $|V|$ and $|E|$ denote the number of vertices and edges respectively.

A unique vertex identifier is assigned to each vertex and a vertex may be associated with a set of attributes describing the properties of the entity represented by the vertex. For presentation convenience, we interchangeably use the terms "vertex attribute" and "vertex state" throughout this chapter. For a weighted graph where each edge has its modifiable, user-defined value, we model each edge weight as an attribute of its source vertex. This allows us to treat all vertices as mutable data and edges as immutable data during iterative graph computations. For instance, when a graph is loaded for PageRank computations and vertex u has its out-edge degree $d(u)$, the graph topology does not change and each of u's out-edges contributes the fixed portion (i.e., $1/d(u)$) to the next round of PageRank values during all the iterations. Thus, we can consider edges immutable for PageRank computations. Similarly for SSSP (Single-Source Shortest Path) computations, the edge weight usually denotes the distance of a road segment and thus is immutable during the computations. This separation between mutable and immutable data by design provides GraphMap an opportunity to employ compact and locality-aware graph storage structure for both in-memory and on-disk placement. Furthermore, since most of large graphs have at least tens or hundreds times more edges than vertices, we can significantly reduce the memory requirement for loading and processing large graphs. We will show in the subsequent sections that by utilizing such a clean separation between mutable and immutable data components in a graph, we can significantly reduce the amount of non-sequential accesses in each iteration for many iterative graph algorithms.

In order to provide access locality-optimized grouping of edges in GraphMap, we categorize all edges connected to a vertex into three groups based on their direction: out-edges, in-edges, and bi-edges.

Definition 18 (Out-edges, in-edges, and bi-edges) Given a graph $G = (V, E)$, the set of **out-edges** of a vertex $v \in V$ is denoted by $E_v^{out} = \{(v, v')|(v, v') \in E\}$. Conversely, the set of **in-edges** of v is denoted by $E_v^{in} = \{(v', v)|(v', v) \in E\}$. We also define **bi-edges** of v as the union of its out-edges and in-edges, denoted by $E_v^{bi} = E_v^{out} \cup E_v^{in}$.

For each graph to be processed by GraphMap, we build a vertex block (VB) for each vertex. A vertex block consists of an *anchor* vertex and its directly connected edges and vertices. Since different graph algorithms have different computation characteristics, in GraphMap, we support three different types of vertex blocks based on the edge direction from the anchor vertex: (1) out-edge vertex block (out-VB), (2) in-edge vertex block (in-VB), and (3) bi-edge vertex block (bi-VB). One may view an out-edge vertex block as a source vertex and its adjacency list via out-edges, i.e., the list of destination vertex IDs connected to the same source vertex via its out-edges. Similarly, an in-edge vertex block can be viewed as a destination vertex and its adjacency list via its in-edges, i.e., the list of source vertex IDs connected to the same destination vertex via its in-edges. We below formally define the concept of vertex block (VB).

Definition 19 (Vertex block) Given a graph $G = (V, E)$ and vertex $v \in V$, the **out-edge vertex block** of vertex v is a subgraph of G, which consists of v as its anchor vertex and all of its out-edges, denoted by $VB_v^{out} = (V_v^{out}, E_v^{out})$ such that $V_v^{out} = \{v\} \cup \{v^{out} | v^{out} \in V, (v, v^{out}) \in E_v^{out}\}$. Similarly, the **in-edge vertex block** of v is defined as $VB_v^{in} = (V_v^{in}, E_v^{in})$ such that $V_v^{in} = \{v\} \cup \{v^{in} | v^{in} \in V, (v^{in}, v) \in E_v^{in}\}$. We define the **bi-edge vertex block** of v as $VB_v^{bi} = (V_v^{bi}, E_v^{bi})$ such that $V_v^{bi} = V_v^{in} \cup V_v^{out}$.

In the subsequent sections we will describe several highlights of GraphMap design.

4.2.2 Two-Phase Graph Partitioning

We design a two-phase graph partitioning algorithm, which performs global hash partitioning followed by local range partitioning at each of the n worker machines in a compute cluster. Hash partitioning on vertex IDs first divides a large graph into a set of vertex blocks (VBs) and then assigns each VB to one worker machine. By using the lightweight global hash partitioning, a large graph can be rapidly distributed across the cluster of n worker machines while ensuring data-level load balance. In order to reduce non-sequential disk accesses at each worker machine, we sort all VBs assigned to each worker machine in the lexical order of their anchor vertex IDs and further partition the set of VBs at each worker machine into r chunks such that VBs are clustered physically by their chunk ID.

The parameter r is chosen such that each range partition (chunk) can fit into the working memory available at the worker machine. The range partitioning is concurrently performed at all the worker machines.

Definition 20 (Hash partitioning) Let $G = (V, E)$ denote an input graph. Let $hash(v)$ be a hash function for partitioning and $VB(v)$ denote the vertex block anchored at vertex v. The hash partitioning P of G is represented by a set of n partitions, denoted by $\{P_1, P_2, \ldots, P_n\}$ such that each partition P_i $(1 \leq i \leq n)$ consists of a set of vertices V_i and a set of VBs B_i such that $V_i = \{v | hash(v) = i, v \in V\}$, $B_i = \{VB(v) | v \in V_i\}$ and $\bigcup_i V_i = V$, $V_i \bigcap V_j = \emptyset$ for $1 \leq i, j \leq n$, $i \neq j$.

In the first prototype implementation of GraphMap, given a cluster of n worker machines, we physically store a graph at n worker machines by hash partitioning and then divide the set of VBs assigned to each of the n worker machines into r range partitions. GraphMap uses the hash partitioning by default for global graph partitioning across the cluster of n worker machines because it is super fast and we do not need to keep any additional data structure to record the partition assignment for each vertex. However, GraphMap can be easily extended to support any other partitioning techniques because its in-memory and on-disk representation is designed to store partition assignments generated by any partitioning techniques, such as Metis [60], ParMetis [44], SHAPE [71], to name a few.

4.2.3 Supporting Vertex-Centric API

Most iterative graph processing systems adopt the "think like a vertex" vertex-centric programming model [80, 45, 77]. To implement an iterative graph algorithm based on the vertex-centric model, users write a vertex-centric program, which defines what each vertex does for *each* iteration of the user-defined iterative graph algorithm, such as PageRank, SSSP and Triangle Counting. In each iteration, vertices of the input graph execute the same vertex program in parallel. A typical vertex program consists of three steps in each iteration: (1) a vertex reads its current value and gathers its neighboring vertices' values, usually along its in-edges. (2) the vertex may update its value based on its current value and

gathered values. (3) if updated, the vertex propagates its updated value to its neighboring vertices, usually along its out-edges.

Each vertex has its transition state flag with either *active* or *inactive*. In each iteration, only active vertices run the vertex program. For some algorithms such as PageRank and Connected Component (CC), every vertex is active in the first iteration and thus all vertices participate in the computation. On the other hand, for some algorithms such as SSSP, only one vertex is active in the first iteration and some vertices may be inactive during all the iterations. A vertex can deactivate itself, usually at the end of an iteration, and can also be reactivated by other vertices. The iterative graph algorithm terminates if all vertices are inactive or an user-defined convergence condition, such as the number of iterations, is satisfied.

Existing distributed iterative graph processing systems provide a mechanism for interaction among vertices, mostly along edges. Pregel [80] employs a pure message passing model in which vertices interact by sending messages along their outgoing edges and, in the current iteration, each vertex receives messages sent by other vertices in the previous iteration. In GraphLab/PowerGraph [45, 77], vertices directly read their neighboring vertices' data through shared state.

One representative category of existing distributed graph processing systems is based on the Bulk Synchronous Parallel (BSP) [114] computation model and the shared-nothing architecture. A typical graph application based on the BSP model starts with an initialization step in which the input graph is read and partitioned/distributed across the worker machines in the cloud. In subsequent iterations, the worker machines compute independently in parallel in each iteration and the iterations are separated by global synchronization barriers in which the worker machines communicate each other to integrate the results from distributed computations performed at different workers. Finally, the graph application finishes by writing down its results.

Algorithm 3 shows an example of the Single-Source Shortest Path (SSSP) algorithm, based on the vertex-centric model and the BSP model, implemented in Apache Hama's graph package, an open-source implementation of Pregel. In iteration (or *superstep*) 0, each

Algorithm 3 SSSP in Apache Hama

```
    compute(messages)
 1: if getSuperstepCount() == 0 then
 2:     setValue(INFINITY);
 3: end if
 4: int minDist = isStartVertex() ? 0 : INFINITY;
 5: for int msg : messages do
 6:     minDist = min(minDist, msg);
 7: end for
 8: if minDist < getValue() then
 9:     setValue(minDist);
10:     for Edge e : getEdges() do
11:         sendMessage(e, minDist+e.getValue())
12:     end for
13: end if
14: voteToHalt();
    combine(messages)
15: return min(messages)
```

vertex sets its vertex value as *infinity* (line 2). In subsequent iterations, each vertex picks the smallest distance among the received messages (line 5-7) and, if the distance is smaller than its current vertex value, the vertex updates its vertex value using the smallest distance (line 9) and propagates the updated distance to all its neighboring vertices along out-edges (line 10-12). At the end of each iteration, it changes its status to *inactive* (line 14). If the vertex receives any message, it will be reactivated in the next iteration and then run the vertex program again. To reduce the number of messages over the network, users can define a combiner, which finds the minimum value of messages for each destination vertex (line 15).

4.2.4 GraphMap Programming API

GraphMap supports two basic programing abstractions at API level: the vertex-centric model, similar to Pregel-like systems, and the VB partition-centric model. Given a vertex-centric program, such as SSSP in Algorithm 3, GraphMap converts it into a GraphMap program, which utilizes the in-memory and on-disk representation of GraphMap and the performance optimizations enabled by GraphMap in terms of access locality and efficient memory consumption (See the next sections for detail).

The VB partition-centric API is provided by GraphMap for advanced users who are familiar with GraphMap's advanced features and the BSP model. Note that, unlike the vertex-centric model, a VB partition-centric program defines what each *VB partition* (a set

Algorithm 4 SSSP in GraphMap

```
    compute(messages)
 1: if getSuperstepCount() == 0 then
 2:     for Vertex v : readAllVertices() do
 3:         if v.isStartVertex() then
 4:             setValue(v, 0);
 5:             setActive(v);
 6:         else
 7:             setValue(v, INFINITY);
 8:         end if
 9:     end for
10: end if
11: for Message msg : messages do
12:     if msg.value < getValue(msg.target) then
13:         setValue(msg.target, msg.value);
14:         setActive(msg.target);
15:     end if
16: end for
17: for Vertex v : readActiveVertices() do
18:     for Edge e : v.getEdges() do
19:         msg = createMessage(e.destination,
                      getValue(v)+e.getValue());
20:         sendMessage(getWorker(e.destination), msg);
21:     end for
22: end for
23: deactivateAll();
```

of VBs) does for *each* iteration. Table 9 shows some core methods provided by GraphMap. Algorithm 4 demonstrates how SSSP is implemented using the VB partition-centric API. We emphasize that we are not claiming that the VB partition-centric API is more concise than the vertex-centric API. Our main goal of the VB partition-centric API is to expose partition-level methods and thus provide more optimization opportunities for advanced users. Recall that users can run their vertex-centric programs on GraphMap as they are without the need to learn the VB partition-centric API. Due to the space constraint, we here omit the further detail on this advanced API design.

Table 9: GraphMap Core Methods

method	description
setValue(vertex, value)	update the value of the vertex
getValue(vertex)	return the value of the vertex
readAllVertices()	return an iterator for all vertices of this partition
readActiveVertices()	return an iterator for all active vertices of this partition
setActive(vertex)	set the vertex as active
createMessage(vertex, value)	create a message including the destination vertex and value
sendMessage(worker, msg)	send the message to the worker
getWorker(vertex)	return the worker which is in charge of the vertex
deactivateAll()	deactivate all vertices of this partition

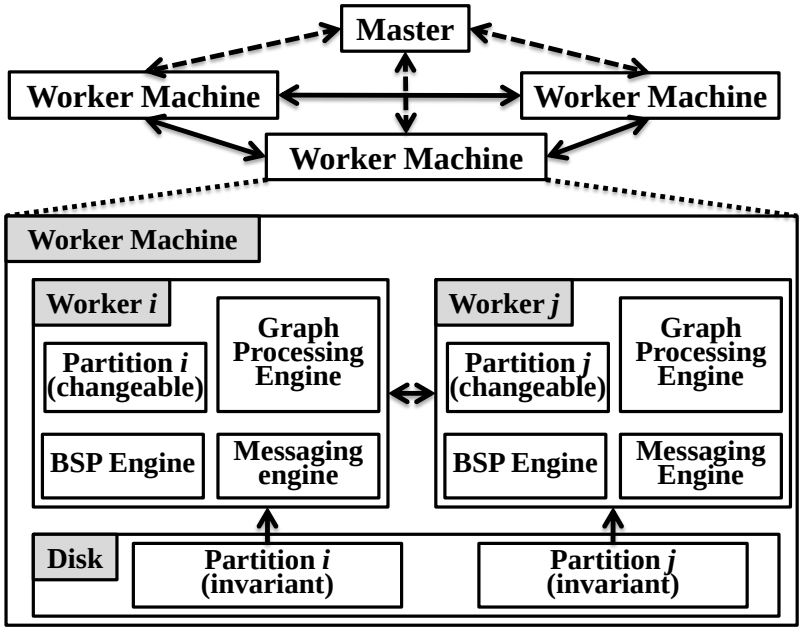


Figure 25: GraphMap System Architecture

4.2.5 System Architecture

Fig. 25 shows the system architecture of GraphMap. Similar to Pregel-like systems, GraphMap supports the BSP model and the message passing model for iterative graph computations. GraphMap consists of a master machine and a set of worker machines. The master accepts graph analysis requests from users and coordinates the worker machines to run the graph algorithms on the input graph datasets. For large graphs, the two phase graph partitioning task is also distributed by the master to its worker machines. The worker machines execute the graph programs by interacting with each other through messages.

Each worker machine can define a set of worker slots for task-level parallelism and each worker is in charge of a single partition. Each worker task keeps the mutable data of its assigned partition in memory (the set of vertices) and in each iteration, it reads the invariant data of the partition from disk (VB blocks) for graph computations and updating the mutable data. In addition, each worker task receives messages from and sends messages to other workers using the messaging engine and enters the global synchronization barrier of the BSP model at the end of each iteration using the BSP engine. We categorize the

messages into two types based on the use of the network: *intra-machine* messages between two workers in the same worker machine and *inter-machine* messages between two workers in the different worker machines. In GraphMap, we bundle a set of messages to be transferred to the same worker at the end of each iteration for batch transmission across workers.

4.3 Locality-based Data Placement

In this section, we introduce our locality-based storage structure for GraphMap. We provide an example to illustrate our in-memory and on-disk representation of graph partitions. In the next section we describe how GraphMap can benefit from the locality-optimized data partitions and data placements to speed up iterative graph computations.

We have mentioned earlier that in most iterative graph algorithms, only vertex data are mutable while edge data are invariant during the entire iterative computations. By cleanly separating graph data into *mutable* and *invariant* (or read-only) data, we can store most or all of the mutable data in memory and access invariant data from disk by minimizing non-sequential IOs. In contrast to existing Pregel-like distributed graph processing systems where each worker machine needs to be able to hold not only the graph data but also their intermediate results and messages in memory, the GraphMap approach promotes the locality-aware integration of external storage with memory-intensive graph computations. The GraphMap design offers two significant advantages: (1) We can considerably reduce the memory requirement for running iterative graph applications by keeping only mutable data in memory and thus enable many more graphs and algorithms to run on GraphMap with respectable performance. (2) By designing a locality-aware data placement strategy such that vertex blocks belonging to the same partition will be stored contiguously on disk, we can speed up the access to the graph data stored on disk through sequential disk IOs for each iteration of the graph computations. For example, we keep all vertices and their values belonging to one partition in memory while keeping all the edges associated with these vertices and the edge property values, if any, on disk. Thus, in the context of vertex blocks in a partition, we maintain only anchor vertices and their values in memory and store all the corresponding vertex blocks contiguously on disk. In each iteration, for each *active*

anchor vertex, we read its vertex block from disk and execute the graph computation in three steps as outlined in Section 4.2.3. For those graphs in which the number of edges is much larger than the number of vertices (e.g., more than two orders of magnitude larger in some real-world graphs), this design can considerably reduce the memory requirement for iterative graph computations even in the presence of long radius and skewed vertex degree distribution, because we do not require keeping edges in memory.

For anchor vertex values, which may be read and updated over the course of the iterative computations, such as the current shortest distance in SSSP and the current PageRank value, we maintain a mapping table that stores the vertex value for each anchor vertex in memory. Since each worker is in charge of one partition in GraphMap, only anchor vertices of its assigned partition are loaded in memory on each worker. For read-only edge data (i.e., vertex blocks of the anchor vertices), we need to carefully design its disk representation because otherwise it would be too costly to load vertex blocks from disk in each iteration. To tackle this challenge, we consider two types of access locality in graph algorithms: 1) edge access locality and 2) vertex access locality. By the *edge access locality*, we mean that all edges (out-edges, in-edges or bi-edges) of an anchor vertex are accessed together to update its vertex value. By using the vertex blocks as our building blocks for storage on disk, we can utilize the edge access locality because all edges of an anchor vertex are placed together. By the *vertex access locality*, we mean that the anchor vertices (and their vertex blocks) of a partition are accessed by the same worker in every iteration. To utilize the vertex access locality, for each partition, we store its all vertex blocks into contiguous disk blocks to utilize sequential disk accesses when we read the vertex blocks from disk in each iteration. In addition to the sequential disk accesses, in order to support efficient random accesses for reading the vertex block of a specific vertex, we store the vertex blocks in sorted order by their anchor vertex identifiers and create an index block that stores the start vertex identifier for each data block. In other words, we use range partitioning in which each data block stores vertex blocks of a specific range.

Fig. 26 shows an example of the in-memory and on-disk graph data representation for a partition held by a worker in GraphMap. All anchor vertices of the partition and their

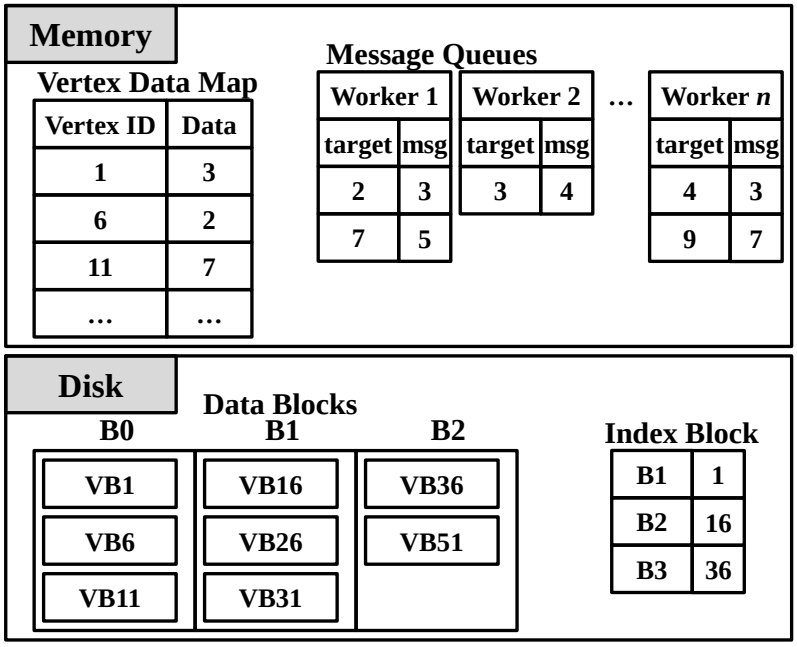


Figure 26: Graph Representation in GraphMap (single worker)

current vertex data are stored in a mapping table in memory. Since GraphMap employs the BSP model based on messaging, we keep an incoming message queue that stores all messages sent to this worker and an outgoing message queue for each worker in memory. On disk, eight vertex blocks are stored in three data blocks and one index block is created to store the start vertex for each data block.

4.4 Locality-based Optimizations

In GraphMap, two levels of parallel computations can be provided: (1) Workers can process graph partitions in parallel; and (2) Within each partition, we can compute multiple vertex blocks concurrently using multi-threading. By combining the graph parallel computations with our locality-based data placement, each parallel task can run independently with minimal non-sequential disk IOs. Since the vertex blocks belonging to the same partition are accessed by the same worker and stored in contiguous disk blocks, we can speed up graph computations in each iteration by combining parallelism with the sequential disk IOs for reading the vertex blocks.

It is interesting to note that different iterative graph algorithms may have different

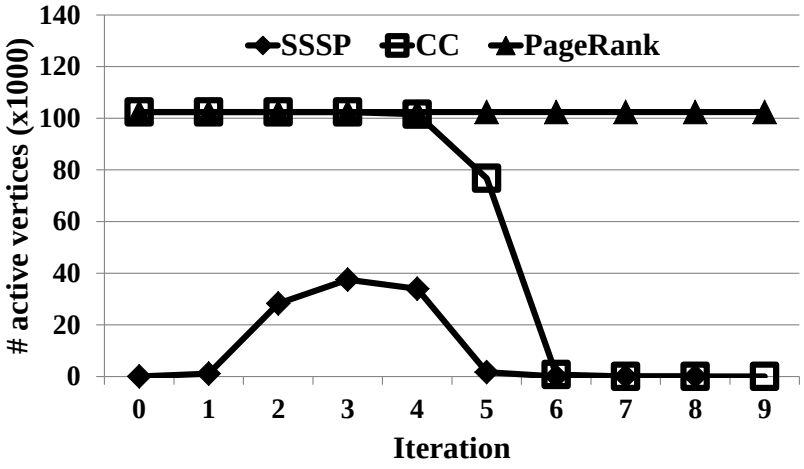


Figure 27: The Number of Active Vertices per Iteration

computation patterns and sequential disk accesses are not efficient for all types of graph computation patterns. For example, Fig. 27 shows the number of active vertices per iteration of a worker for three different iterative graph algorithms, Single-Source Shortest Path (SSSP), Connected Components (CC), and PageRank, on the orkut graph [83]. In PageRank, since all vertices are always active during all iterations and thus all vertex blocks of the anchor vertices are required in each iteration, our sequential disk accesses would be always efficient for reading the vertex blocks. On the other hand, in SSSP and CC, the number of active vertices is different for different iterations. When the number of active vertices is small, using the sequential accesses and reading the vertex blocks of all anchor vertices would be far from optimal because we do not need to evaluate the vertex blocks of most anchor vertices.

Based on this observation, we develop another locality-based optimization in GraphMap, which can dynamically choose between the sequential disk accesses and the random disk accesses based on the computation loads of the current iteration for each worker. This dynamic locality-based adaptation enables us not only to progressively filter out non-active vertices in each of the iterations for the iterative graph algorithms but also to avoid unnecessary and thus wasteful sequential disk IOs as early as possible. Recall that we store the vertex blocks of a partition in sorted order by their vertex identifiers and create an index block to support efficient random accesses. Given a query vertex, we need one (when

the index block resides in memory) or two (when the index block is not in memory) disk accesses to read its vertex block. Specifically, in each iteration of a worker, if the number of active vertices is less than a system-defined (and user-modifiable) threshold θ, we choose the random disk accesses as our access method and read the index block from disk first, if not in memory. Based on the block information (i.e., start vertex for each data block) stored in the index block, we select and read only those data blocks that include the vertex block of active vertices. If the number of active vertices is equal to or larger than θ, we choose the sequential disk accesses and read the vertex blocks of all anchor vertices regardless of the current active vertices.

Because different clusters and different worker machines have different disk access performance, we also dynamically change the value of θ for each worker machine. Conceptually, by monitoring the current processing time of each random disk access and one full scan (i.e., sequential disk accesses for reading all vertex blocks), we calculate the break-even point, in which one full scan time is equal to the time of r random disk accesses, and use r as the value of θ.

The algorithm of updating the value of θ is formally defined as follows. Let θ_{iw}, s_{iw}, r_{iw}, and a_{iw} denote the threshold, one full scan time, total random disk access time, and the number of active vertices in iteration i on worker w respectively. If the full scan (or random disk access) is not used in iteration i on worker w, s_{iw} (or r_{iw}) is not defined (i.e., not a valid number). We use m and n, initially having 0 (zero), to denote the IDs of the last iteration where the full scan and random disk access was used respectively. θ_{0w} is the initial threshold on worker w and calculated empirically (e.g., random disk access time for 2% of all anchor vertices is similar to sequential disk access time for all anchor vertices on worker w). In iteration i ($i > 0$), before running the vertex program for each active vertex, we calculate the new threshold as follows:

$$\theta_{iw} = \begin{cases} \theta_{(i-1)w}, & \text{if } m = 0 \text{ or } n = 0 \\ s_{mw}\frac{a_{nw}}{r_{nw}}, & \text{otherwise.} \end{cases}$$

In addition, when we store the vertex block of each anchor vertex in a data block, we

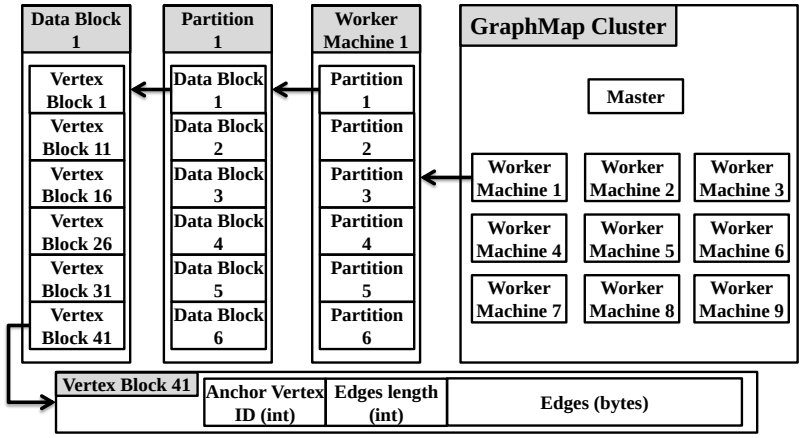


Figure 28: Hierarchical Disk Representation in GraphMap

bundle all out-edges (or in-edges or bi-edges) of the anchor vertex and store them together, as shown in Fig. 28, to utilize the edge access locality.

4.5 Experimental Evaluation

In this section, we report the experimental evaluation results of the first prototype of GraphMap for various real-world graphs and iterative graph algorithms. We first explain the characteristics of graphs we used for our evaluation and the experimental settings. We categorize the experimental results into four sets: 1) We show the execution time of various iterative graph algorithms in GraphMap and compare it with that of an Pregel-like system. 2) We present the effects of our dynamic access methods for various graph datasets. 3) We evaluate the scalability of GraphMap by increasing the number of workers in the cluster. 4) We compare GraphMap with other state-of-the-art graph systems.

4.5.1 Datasets and Graph Algorithms

We evaluate the performance of GraphMap using real-world graphs of different sizes and different characteristics for three types of iterative graph algorithms. Table 10 gives a summary of the datasets used for our evaluation. The first type of graph algorithms is represented by PageRank. In these algorithms, all vertices are always active during all iterations. The second type of graph algorithms is represented by Connected Components (CC), in which all vertices of the graph are active in the first iteration and then the number

of active vertices starts to decrease as the computation progresses towards convergence. The third type of graph algorithms is represented by Single-Source Shortest Path (SSSP), where only the start vertex is active in the first iteration and the number of active vertices increases in early iterations and decreases in later iterations. We choose these three types of graph applications because they display different computation characteristics as we have shown earlier in Fig. 27.

Table 10: Datasets (GraphMap)

Dataset	#vertices	#edges
hollywood-2011 [33]	2.2M	229M
orkut [83]	3.1M	224M
cit-Patents [72]	3.8M	16.5M
soc-LiveJournal1 [26]	4.8M	69M
uk-2005 [33]	39M	936M
twitter [66]	42M	1.5B

4.5.2 Setup and Implementation

We use a cluster of 21 machines (one master and 20 worker machines) on Emulab [115]: each node is equipped with 12GB RAM, one quad-core Intel Xeon E5530 processor, and two 7200 rpm SATA disks (500GB and 250GB), running CentOS 5.5. They are connected via a 1 GigE network. We run three workers on each worker machine and each worker is a JVM process with a maximum heap size of 3GB, unless otherwise noted. When we measure the computation time, we perform five runs under the same setting and show the *fastest* time to remove any possible bias posed by OS and/or network activity.

In order to compare with distributed memory-based graph systems, we use Apache Hama (Version 0.6.3), an open source implementation of Pregel. Another reason we choose Hama is that we implement the BSP engine and the messaging engine of GraphMap workers by adapting the BSP module and the messaging module of Apache Hama for the first prototype of GraphMap. This allows us to compare GraphMap with Hama's graph package more fairly.

To implement our vertex block-based data representation on disk, we utilize Apache HBase (Version 0.96), an open source wide column store (or two-dimensional key-value store), on top of Hadoop Distributed File System (HDFS) of Hadoop (Version 1.0.4). We

Table 11: Total Execution Time and the Number of Messages

Dataset	Total execution time (sec)					
	SSSP		CC		PageRank	
	Hama	GraphMap	Hama	GraphMap	Hama	GraphMap
hollywood-2011	108.776	18.347	177.854	39.365	268.474	111.466
orkut	108.744	21.345	195.841	54.383	286.054	111.46
cit-Patents	27.693	12.337	24.646	12.335	30.688	18.353
soc-LiveJournal1	48.697	18.346	60.734	33.357	75.76	39.369
uk-2005	Fail	156.49	Fail	706.329	Fail	573.964
twitter	Fail	150.486	Fail	303.653	Fail	1492.966
	The number of messages					
hollywood-2011	229M	80M	1.2B	348M	2.1B	2.1B
orkut	224M	123M	1.3B	548M	2.0B	2.0B
cit-Patents	219K	212K	17M	15M	149M	149M
soc-LiveJournal1	68M	51M	359M	243M	616M	616M
uk-2005	Fail	450M	Fail	5.2B	Fail	8.3B
twitter	Fail	585M	Fail	1.5B	Fail	13.2B

choose HBase for the first prototype of GraphMap because it has several advantages. First, it provides a fault-tolerant way of storing graph data in the cloud. Since HBase utilizes the data replication of HDFS for fault-tolerance, GraphMap will continue to work even though some worker machines fail to perform correctly. Second, since HBase row keys are in sorted order and adjacent rows are usually stored in the same HDFS block (a single file in the file system), we can directly utilize HBase's range scans for implementing sequential disk accesses. Third, we can place all the vertex blocks of a partition in the same worker machine (called a region server) by using the HBase regions and renaming vertex identifiers. Specifically, we first pre-split the HBase table for the input graph into a set of regions in which each region is in charge of one hash partition. Next, we rename each vertex identifier by adding its partition identifier as a prefix of its new vertex identifier, such as "11-341" in which "11" and "341" represent the partition identifier and the original vertex identifier respectively. Thus all vertex blocks of a partition are stored in the same region. In other words, our hash partitioning is implemented by renaming vertex identifiers and using the pre-split regions and our range partitioning on each partition is implemented by HBase, which stores rows in sorted order by their identifier. Fourth, to implement our edge access locality-based approach, we bundle all edges of a vertex block and store the bundled data in a single column because the data is stored together on disk. Another possible technique is to use a column for each edge of a vertex block using the same column family because all column family members are stored together on disk by HBase. However, to eliminate

the overhead of handling many columns, we implement the former technique for our edge access locality-based approach.

4.5.3 Iterative Graph Computations

We first compare the total execution time of GraphMap with that of Hama's graph package for the three iterative graph algorithms. Table 11 shows the results for different real-world graphs. For SSSP, we report the execution time when we choose the vertex having the largest number of out-edges as the start vertex except the uk-2005 graph in which we choose the vertex having the third largest number of out-edges because only about 0.01% vertices are reachable from each of the top two vertices. For PageRank, we report the execution time of 10 iterations. The result clearly shows that GraphMap outperforms Hama significantly on all datasets for all algorithms (PageRank, SSSP and CC). For large graphs such as uk-2005 and twitter, GraphMap considerably reduces the memory requirement for iterative graph algorithms. However, Hama fails for all algorithms because it needs not only to load all vertices and edges of the input graph but also to hold all intermediate results and messages in memory. Thus Hama cannot handle those large graphs, such as uk-2005 (936M edges) and twitter (1.5B edges) datasets, where the number of edges is approaching or exceeding one billion.

GraphMap not only reduces the memory requirement for iterative graph algorithms but also significantly improves the iterative graph computation performance compared to Hama. For SSSP, CC and PageRank, GraphMap is 2x-6x, 1.8x-4.5x and 1.7x-2.6x faster than Hama respectively. Given that both GraphMap and Hama use the same messaging and BSP engines, the difference in terms of the number of messages in Table 11 is due to the effect of the combiner. The GraphMap's messages are counted after the combiner is executed and, on the other hand, Hama reports only the numbers, which are measured before the combiner is executed. For PageRank, the number of messages is almost the same for both systems because no combiner is used.

To provide in-depth analysis, we further divide the total execution time into the vertex processing time and the synchronization (global barrier) time per iteration as shown in

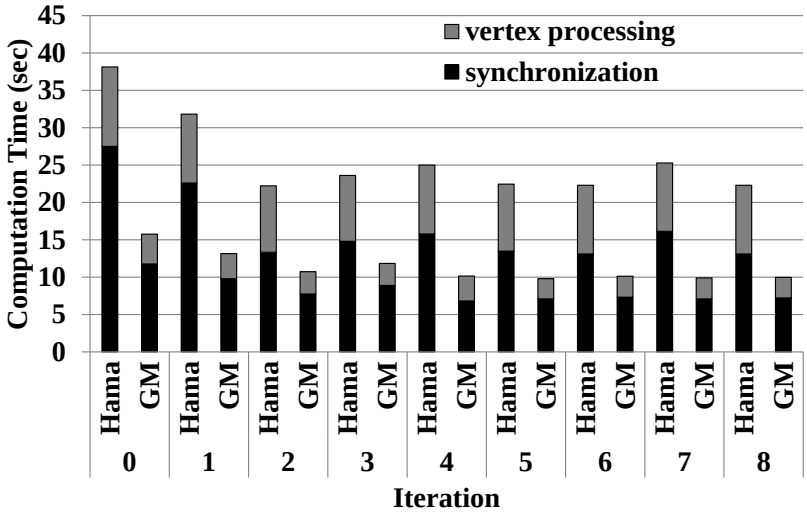
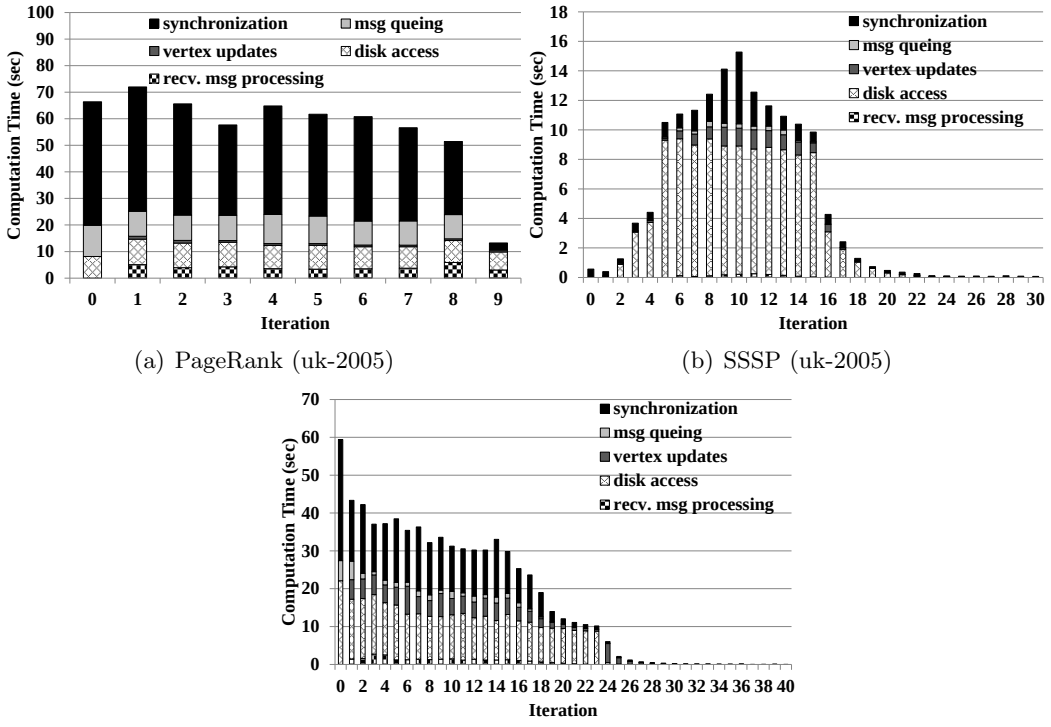


Figure 29: Comparison with Hama (PageRank on orkut)



(a) PageRank (uk-2005)

(b) SSSP (uk-2005)

(c) CC (uk-2005)

Figure 30: Breakdown of Execution Time per Iteration (single worker)

Table 12: Effects of Dynamic Access Methods

Dataset	Total execution time (sec)					
	SSSP			CC		
	GraphMap-Sequential	GraphMap-Random	GraphMap-Dynamic	GraphMap-Sequential	GraphMap-Random	GraphMap-Dynamic
hollywood-2011	24.354	27.345	18.347	45.383	81.405	39.365
orkut	27.35	33.356	21.345	57.384	126.455	54.383
cit-Patents	15.34	12.34	12.337	18.34	12.332	12.335
soc-LiveJournal1	24.348	36.357	18.346	36.361	120.447	33.357
uk-2005	1225.637	225.555	156.49	2033.522	2898.407	706.329
twitter	252.598	267.622	150.486	712.085	721.089	303.653

Fig. 29. The synchronization time includes not only the message transfer time among workers but also the waiting time until the other workers complete their processing in the current iteration. The vertex processing time includes the vertex update time (the core part defined in the vertex-centric program), received message processing time and message queuing time for messages to be sent during the next synchronization. For GraphMap, it also includes the disk (HBase) access time. It is interesting to note that, even though Hama is the in-memory system, its vertex processing time is much slower than that of GraphMap, which accesses HBase to read the vertex blocks stored on disk, for all iterations. This result shows that a carefully designed framework based on the access locality of iterative graph algorithms can be competitive with and in some cases outperform the in-memory framework in terms of the total execution time even though it requires disk IOs in each iteration for reading a part of graph data.

We split the vertex processing time of GraphMap for further details as shown in Fig. 30. We could not find measurement points to gather such numbers for Hama. For PageRank, all iterations have similar results except the first iteration, in which there is no received message, and the last iteration, in which no message is sent. Note that the vertex update time, the core component for the vertex-centric model, is only a small part in the total execution time. For SSSP, the disk IOs from iteration 5 to iteration 15 are almost the same because GraphMap chooses the sequential accesses based on our dynamic access methods. From iteration 16 to iteration 30, the disk IOs continue to drop until the algorithm converges thanks to our dynamic locality-based adaption.

4.5.4 Effects of Dynamic Access Methods

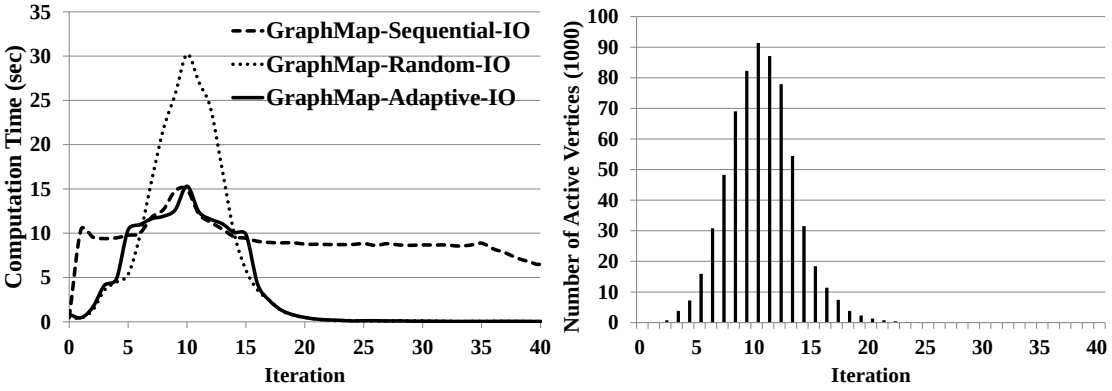
Table 12 shows the effects of the dynamic access methods of GraphMap, compared to two baseline approaches that use only sequential accesses or only random accesses in all iterations for SSSP and CC. For PageRank, GraphMap always chooses the sequential accesses because all vertices are active during all iterations. The experimental results clearly show that GraphMap with the dynamic access methods offers the best performance because it chooses the optimal access method for each worker and in each iteration based on the current computation loads, such as the ratio of active vertices to total vertices in a partition.

Table 12 also shows that for the cit-Patents graph dataset, GraphMap always chooses the random accesses because only 3.3% vertices are reachable from the start vertex and thus the number of active vertices in each iteration is very small. For SSSP on the uk-2005 graph, the baseline approach using only sequential accesses is 8x slower than GraphMap. This is because it takes 198 iterations for SSSP on the uk-2005 graph to converge and the baseline approach always runs with the sequential disk accesses even though the number of active vertices is very small in most iterations.

Fig. 31 shows the effects of GraphMap's dynamic access methods per iteration, for the first 40 iterations, on a single worker using the uk-2005 graph. The result shows that GraphMap chooses the optimal access method in most of the iterations based on the number of active vertices. It is interesting to note that GraphMap chooses the sequential accesses in iteration 5 and 15 even though random accesses are faster. This indicates that GraphMap's performance can be improved further by fine-tuning the threshold θ value. In these experiments, θ is empirically set to 2% of all vertices in each partition.

4.5.5 Scalability

To evaluate the scalability of GraphMap framework, we report the SSSP execution results with varying numbers of workers from 60 to 180 using the same cluster, as shown in Table 13. For this set of experiments, we use 1GB as the maximum heap size of each worker for both GraphMap and Hama. The results show that GraphMap needs fewer workers than Hama to run the same graph because it reduces the memory requirement of graph computations.



(a) Computation Time (b) #active vertices

Figure 31: Effects of Dynamic Access Methods

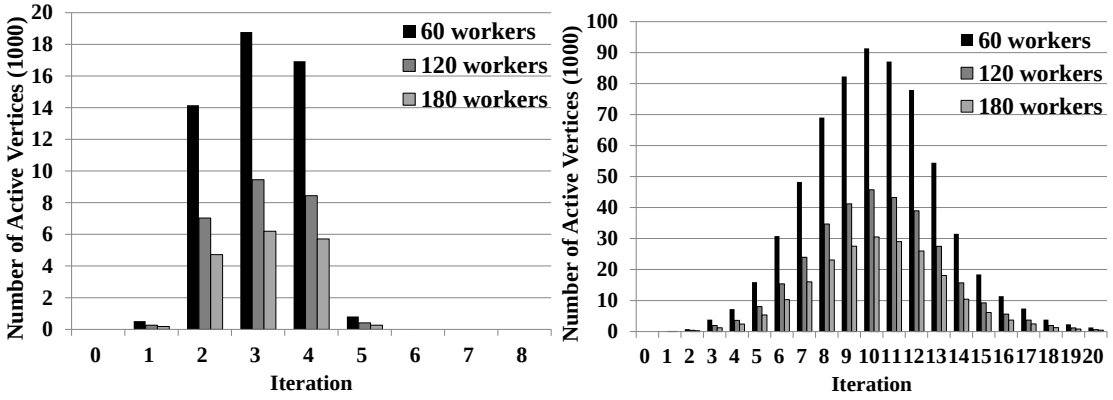
If we run more workers, each worker handles fewer active vertices proportionally, as shown in Fig. 32(a) and Fig. 32(b), because the worker is in charge of a smaller partition. However, by increasing the number of workers, the cost of inter-worker communication will increase significantly, which hurt the computation time even with a smaller number of active vertices on each worker. As shown in Fig. 32(c) and Fig. 32(d), the vertex update time reduces as we increase the number of workers but at the cost of increased synchronization time for coordinating more workers.

Table 13: Scalability (SSSP)

Total execution time (sec)				
		#Workers		
Dataset	**Framework**	**60**	**120**	**180**
hollywood-2011	Hama	Fail	114.801	114.926
	GraphMap	18.352	21.351	27.356
orkut	Hama	Fail	99.784	102.883
	GraphMap	21.36	24.359	30.355
cit-Patents	Hama	27.678	39.738	54.799
	GraphMap	9.348	15.369	18.348
soc-LiveJournal1	Hama	45.683	54.736	75.837
	GraphMap	18.357	21.368	27.356
uk-2005	Hama	Fail	Fail	415.239
	GraphMap	159.517	135.486	138.481
twitter	Hama	Fail	Fail	Fail
	GraphMap	Fail	141.485	126.468

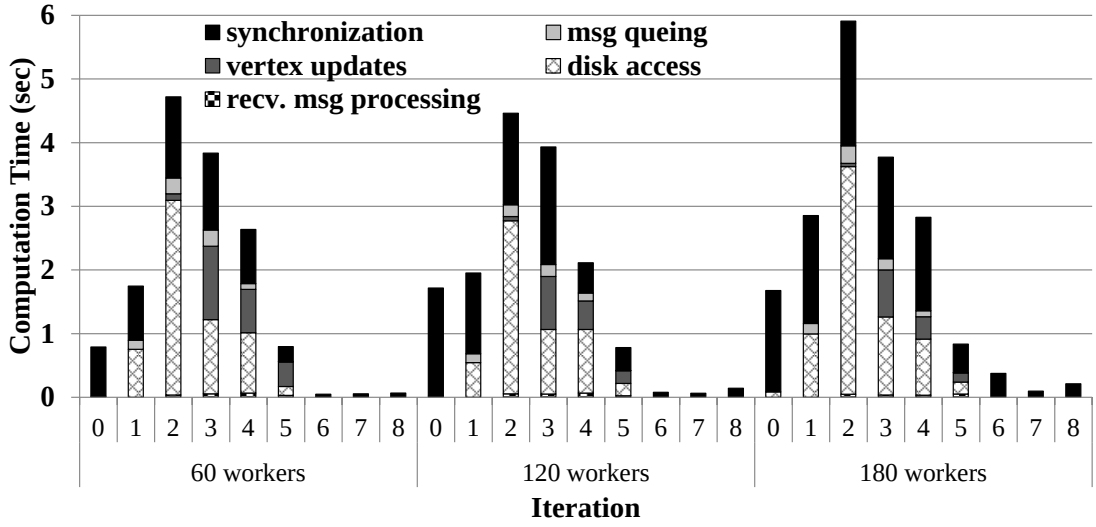
4.5.6 Comparison with Existing Systems

In this section we compare GraphMap with existing representative in-memory graph systems, including GraphX, GraphLab (PowerGraph), Giraph, Giraph++ (with hash partitioning), and Hama, in Table 14. We compare the performance of PageRank and CC on

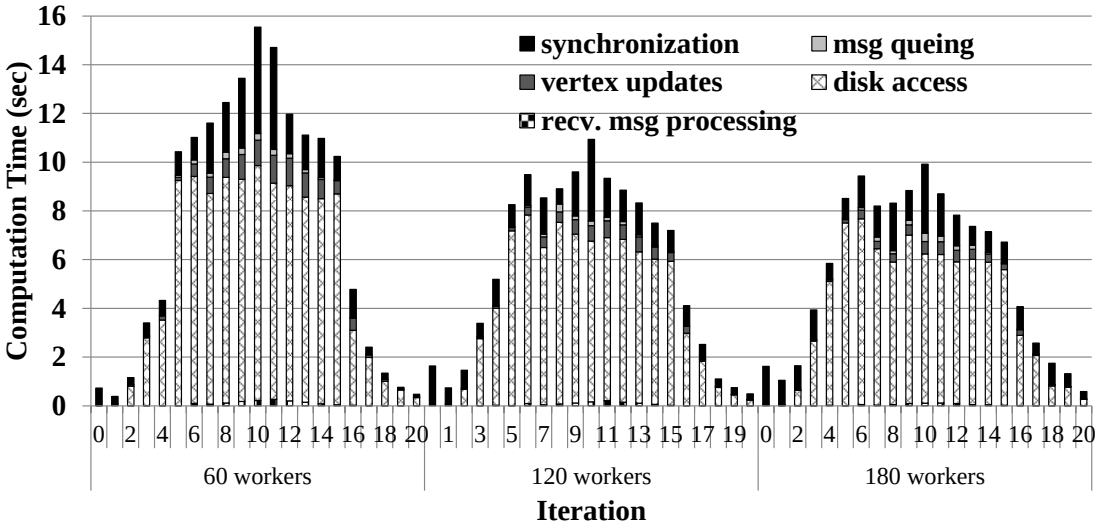


(a) #active vertices (orkut)

(b) #active vertices (uk-2005)



(c) computation time (orkut)



(d) computation time (uk-2005)

Figure 32: Scalability with Varying the Number of Workers

Table 14: System Comparison

System	Setting	CC (sec)		PageRank (sec/iteration)		Type
		twitter	uk-2005 (*uk-2007)	twitter	uk-2005 (*uk-2007)	
GraphMap on Hadoop	21 nodes (21x4=84 cores, 21x12=252GB RAM)	304	706	149	57	Out-of-core
Hama on Hadoop	21 nodes (21x4=84 cores 21x12=252GB RAM)	Fail	Fail	Fail	Fail	In-memory
GraphX on Spark	16 nodes (16x8=128 cores 16x68=1088GB RAM)	251	800*	21	23*	In-memory
GraphLab 2.2 (PowerGraph)	16 nodes (16x8=128 cores 16x68=1088GB RAM)	244	714*	12	42*	In-memory
Giraph 1.1 on Hadoop	16 nodes (16x8=128 cores 16x68=1088GB RAM)	200	Fail*	30	62*	In-memory
Giraph++ on Hadoop	10 nodes (10x8=80 cores 10x32=320GB RAM)	No result reported	723	No result reported	89	In-memory

twitter and uk-2005/uk-2007 datasets. Given that GraphX requires Spark and larger memory to run, we extract the performance results of GraphX, GraphLab and Giraph from [46], annotated with their respective system configurations for the same graph datasets. The results of Giraph++ are extracted from [120]. We offer a number of observations.

First, the testbeds for other systems have larger RAM and a larger number of CPU cores. For example, GraphX, GraphLab, and Giraph run on a cluster with 1,088GB RAM and 128 cores while GraphMap runs on a cluster with 256GB RAM and 84 cores. For CC on the twitter dataset, GraphMap shows comparable performance to these state-of-the-art in-memory graph systems, even though GraphMap uses much less computing resources (less than two-thirds of cores and less than one-fourth of RAM). For example, GraphMap is only 20% slower than GraphX with a much more powerful cluster. GraphMap is even faster than Giraph++ on the uk-2005 dataset. Through our dynamic access methods, GraphMap achieves competitive performance for CC even though it accesses the disk for each iteration. For PageRank, GraphMap is slower than GraphX, GraphLab, and Giraph because it reads all the edge data from disk in each iteration with only two-thirds of CPU cores. This comparison demonstrates the effectiveness of the GraphMap approach to iterative computations of large graphs.

4.6 Related Work

We classify existing systems for iterative graph algorithms into two categories. The first category is the distributed solution that runs the iterative graph computations using a cluster

of commodity machines, represented by distributed memory-based systems like Pregel. The second category of graph systems is the disk-based solution on a single machine, represented by GraphChi [67] and X-Stream [98].

Distributed memory-based systems typically require to load the whole input graph in memory and to have sufficient memory to store all intermediate results and all messages in order to run iterative graph algorithms [80, 45, 77, 106, 46]. Apache Hama and Giraph are the popular open-source implementations of Pregel. GraphX [46] and Pregelix [34] implement a graph processing engine on top of a general-purpose distributed dataflow framework. They represent the graph data as tables and then use database-style query execution techniques to run iterative graph algorithms.

Unlike Pregel-like distributed graph systems, GraphLab [77] and PowerGraph [45] replicate a set of vertices and edges using a concept of ghosts and mirrors respectively. GraphLab is based on an asynchronous model of computations and PowerGraph can run vertex-centric programs both synchronously and asynchronously. Trinity [106] handles both online and offline graph computations using a distributed memory cloud (an in-memory key-value store). In addition, several techniques for improving the distributed memory-based graph systems have been explored, such as dynamic workload balancing [100, 65] and graph-centric view [113].

Disk-based systems focus on improving the performance of iterative computations on a single machine [67, 98, 49, 120, 124]. GraphChi [67] is based on the vertex-centric model. It improves disk access efficiency by dividing a large graph into small shards, and uses a parallel sliding window-based method to access graph data on disk. Unlike the vertex-centric model, X-Stream [98] proposes an edge-centric model to utilize sequential disk accesses. It partitions a large graph into multiple streaming partitions and loads a streaming partition in main memory to avoid random disk accesses to vertices. PathGraph [120] proposes a path-centric model to improve the memory and disk access locality. TurboGraph [49] and FlashGraph [124], based on SSDs, improve the performance by exploiting I/O parallelism and overlapping computations with I/O.

Even though some systems, including Giraph and Pregelix, provide out-of-core capabilities to utilize external memory for handling large graphs, they typically focus only on reducing the memory requirement, not the performance of iterative graph computations. To the best of our knowledge, GraphMap is the first distributed graph processing system, which incorporates external storage into the system design for efficient processing of iterative graph algorithms in a cluster of compute nodes.

4.7 Conclusion

We have presented GraphMap, a distributed iterative graph computation framework, which effectively utilizes secondary storage to maximize access locality and speed up distributed iterative graph computations. This chapter makes three unique contributions. First, we advocate a clean separation of those data states that are mutable during iterative computations from those that are read-only in all iterations. This allows us to develop locality-optimized data placement and data partitioning methods to maximize sequential accesses and minimize random accesses. Second, we devise a two-level graph partitioning algorithm to enable balanced workloads and locality-optimized data placement. In addition, we propose a suite of locality-based optimizations to improve computation efficiency. We evaluate GraphMap through extensive experiments on several real graphs and show that GraphMap outperforms an existing distributed memory-based system for various iterative graph algorithms.

CHAPTER V

ROADALARM: ROAD NETWORK-AWARE SPATIAL ALARMS

Road network-aware spatial alarms extend the concept of time-based alarms to spatial dimension and remind us when we travel on spatially constrained road networks and enter some predefined locations of interest in the future. This chapter argues that road network-aware spatial alarms need to be processed by taking into account spatial constraints on road networks and mobility patterns of mobile subscribers. We show that the Euclidian distance-based spatial alarm processing techniques tend to incur high client energy consumption due to unnecessarily frequent client wakeups. We design and develop a road network-aware spatial alarm processing system, called RoadAlarm, with three unique features. First, we introduce the concept of road network-based spatial alarms using road network distance measures. Instead of using a rectangular region, a road network-aware spatial alarm is a star-like subgraph with an alarm target as the center of the star and border points as the scope of the alarm region. Second, we describe a baseline approach for spatial alarm processing by exploiting two types of filters. We use subscription filter and Euclidean lower bound filter to reduce the amount of shortest path computations required in both computing alarm hibernation time and performing alarm checks at the server. Last but not the least, we develop a suite of optimization techniques using motion-aware filters, which enable us to further increase the hibernation time of mobile clients and reduce the frequency of wakeups and alarm checks, while ensuring high accuracy of spatial alarm processing. Our experimental results show that the road network-aware spatial alarm processing significantly outperforms existing Euclidean space-based approaches, in terms of both the number of wakeups and the hibernation time at mobile clients and the number of alarm checks at the server.

5.1 Introduction

Most of us use time-based alarms almost everyday to remind us the arrival of some predefined time points of interest in the future, such as getting up in the morning. Spatial alarms extend the concept of time-based alarms to spatial dimension and remind us when we enter some predefined locations of interest in the future. An example of spatial alarms is "alert me when I am within 2 miles of the dry clean store at Atlantic Station." Spatial alarms are basic build blocks for many location-based services, such as location-based advertisements, factory danger zone alert system, and sex offender monitoring system. Since the number of smart devices including smart-phones and tablets is rising steeply (The worldwide smart device shipments will reach 2.1 billion units in 2017 [12]), scalable processing of spatial alarms is becoming increasingly important in mobile applications and location-aware computing.

A spatial alarm is defined by four components: a focal point representing the alarm target, a spatial distance representing the alarm region, an owner (or a publisher) of the alarm, and a set of alarm subscribers. Spatial alarms are categorized into three groups by their ownership: *private*, *shared*, and *public*. A *private* alarm has only one subscriber who is also the publisher of the alarm. A *shared* alarm has a publisher and several subscribers approved by the publisher. In terms of a *public* alarm, its publisher does not set any restriction on subscribers and thus anyone can be a subscriber of the alarm. Public alarms are typically classified by alarm interests, such as traffic alerts and coupons from grocery stores and restaurants. Spatial alarms can also be categorized by the motion behavior of their subscribers and their monitoring targets: *moving objects with static targets*, *static objects with moving targets*, and *moving objects with moving targets*. Typical examples of spatial alarms having *moving objects with static targets* are "alert me when I am within 5 miles of a Whole Foods Market in Buckhead" (private) and "notify anyone entering I85 North from Spaghetti Jct. in Atlanta" (public). "The Macy's store at Lenox Square sends advertisements to its customers who are within 10 miles of its store location" (i.e., Macy's customers are spatial alarm targets for the Macy's store and the store will be notified when its customers are within a specified spatial range from the store) is an example of spatial alarms having *static objects with moving targets*. "Alert Lucy when her car is 1 mile apart

from her friends' vehicles on the way to Walt Disney World in Orlando" is an example of *moving objects with moving targets*.

Spatial alarms are essential for many location-based services. Negligent management of spatial alarms can lead to excessive energy consumption of mobile devices, especially those with limited battery power since continuous tracking of mobile devices is known to be costly. For example, according to [16], minimizing use of location services is listed as one tip to extend smart-phones' battery life. Furthermore, the performance of spatial alarm processing can be affected by a number of factors, such as *frequency of wakeups* − how often mobile devices should wake up because of possible alarm hits and *frequency of alarm checks* − how many spatial alarms should be evaluated at each wakeup. Since frequent and possibly unnecessary wakeups and alarm checks not only reduce battery life of mobile devices considerably but also increase the loads of a spatial alarm processing server, we need efficient spatial alarm processing that can reduce the number of unnecessary wakeups and alarm checks at each wakeup. Furthermore, the spatial alarm processing system should scale to a large number of spatial alarms and mobile users while meeting the *high accuracy* goal by minimizing the alarm miss rate.

Existing approaches on spatial alarm processing can be categorized into two groups by their criteria for controlling the frequency of wakeups: *time-based* approaches (e.g., periodic wakeups) and *distance-based* approaches (e.g., safe period [28, 85] and safe region [27, 41]). To the best of our knowledge, no existing research has taken into consideration spatial constraints for traveling on road networks in optimizing spatial alarm evaluation. We argue that although existing approaches can be applied to road network-aware spatial alarms, they fail to incorporate road network distance into spatial alarm definition and spatial alarm processing. As a result, existing approaches tend to incur unnecessary wakeups and shorter hibernation time at mobile clients and unnecessary computation and alarm checks at the spatial alarm processing server.

In this chapter, we present RoadAlarm − a road network-aware spatial alarm processing system. By taking into account spatial constraints on road networks and mobility

patterns of mobile subscribers, RoadAlarm can reduce the frequency of wakeups and increase hibernation time of mobile clients and, at the same time, minimize the computation cost of alarm checks by filtering out those spatial alarms that are irrelevant or far away from the current location of their mobile subscribers. Concretely, we define road network-aware spatial alarms using network distances (e.g., segment length-based or travel time-based). Instead of using a rectangular region, a road network-aware spatial alarm is defined as a star-like subgraph with an alarm target as the center of the star. We define the scope of an alarm region by the set of border points of the star. In addition, we formulate our baseline approach to road network-aware spatial alarm processing by exploiting subscription filtering and Euclidean lower bound filtering. The former can filter out those spatial alarms that are clearly irrelevant by considering only subscribed spatial alarms. The latter can reduce the number of the network distance computations without loss of accuracy. Furthermore, we develop a suite of motion-aware filters as optimization techniques to further reduce the frequency of wakeups as well as the frequency of alarm checks while ensuring high accuracy by considering mobility patterns of mobile subscribers. To the best of our knowledge, RoadAlarm is the first systematic approach to exploring road network-aware and motion-aware filters to reduce the search space and computation cost of road network-aware alarm processing. Our experimental results show that RoadAlarm outperforms existing Euclidean distance-based techniques and can scale to a large and growing number of spatial alarms as well as mobile subscribers.

The rest of the chapter is organized as follows. We give an overview of the RoadAlarm system architecture and define the road network model and road network-aware spatial alarms in Section 5.2. In Section 5.3, we first describe limitations of applying Euclidean distance-based approaches and the Dijkstra's network expansion approach for processing road network-aware spatial alarms. Then we present our baseline approach that utilizes two types of alarm filters to achieve the desired system scalability while maintaining high accuracy. To further optimize the performance of the baseline approach in RoadAlarm, we develop a suite of optimization techniques using four types of motion-aware filters in Section 5.4. We evaluate the performance of RoadAlarm in Section 5.5, outline the

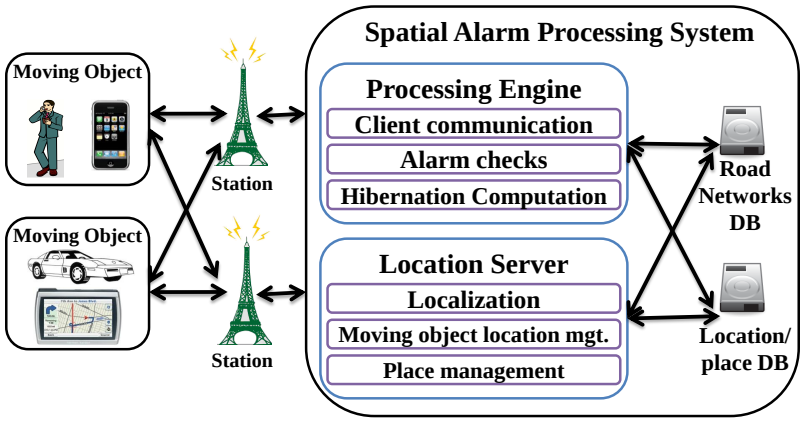


Figure 33: RoadAlarm System Architecture

related work in Section 5.6, and conclude the chapter in Section 5.7.

5.2 Overview

In this section, we describe the system architecture of RoadAlarm and define road network-aware spatial alarms, alarm miss, and hibernation time. A spatial alarm system typically consists of a spatial alarm processing engine and a location server where the locations of moving objects (mobile clients) and the locations of static objects (such as gas stations, restaurants, and so on) are managed. The spatial alarm processing engine communicates with the location server to obtain the current road network locations of mobile subscribers as well as the road network locations of alarm targets for all alarms maintained in its database. The location server uses localization techniques (such as GPS, WiFi or any hybrid localization technology) to keep track of the current positions of moving objects. Fig. 33 presents a sketch of the RoadAlarm system architecture.

We assume that moving objects can be any devices (e.g., smart-phones, tablets, navigation systems) with any localization technology such as GPS and WiFi localization. RoadAlarm adopts the client-server architecture for spatial alarm processing. Concretely, mobile objects may install (publish) their spatial alarms at the location server as private, shared or pubic alarms. In addition to their own private alarms, mobile objects can subscribe to any public alarms of interests and a subset of shared alarms authorized by other alarm owners. Mobile objects need to install the thin client of RoadAlarm as a mobile application on

their devices. Each mobile subscriber will obtain an initial hibernation time at the commit of her alarm subscription. Upon the expiration of its old hibernation time, the mobile client will automatically contact the alarm server on behalf of the mobile subscriber to obtain its new hibernation time. We assume that the mobile clients are able to communicate with the server through wireless data channel. During the hibernation time, the RoadAlarm application is hibernated at the mobile client, and the alarm server consumes zero alarm processing cost for this mobile client.

5.2.1 Road Network Model

A road network is represented by a directed graph $G = (\mathcal{V}, \mathcal{E})$, composed of the road junction nodes $\mathcal{V} = \{n_0, n_1, \ldots, n_N\}$ and directed edges $\mathcal{E} = \{n_i n_j | n_i, n_j \in \mathcal{V}\}$. We refer to an edge $n_i n_j$ as a road segment connecting the two road junction nodes n_i and n_j with direction from n_i to n_j. When a road segment is bidirectional, we use edge $n_i n_j$ and edge $n_j n_i$ to denote the two directions of the same road segment with n_i and n_j as the starting nodes respectively. For each road segment, road-related information can be maintained, such as segment length (e.g., 1.2 miles), speed limit (e.g., 55 mph), current traffic data (e.g., average speed is 35 mph), direction (e.g., one-way road), etc. The length and speed limit of a road segment $n_i n_j$ are denoted by $seglength(n_i n_j)$ in miles and $speedlimit(n_i n_j)$ in miles per hour respectively. Other road-related information such as direction and current traffic data, if available, can be easily incorporated to provide more accurate travel time.

Let n_1 and n_2 denote two road junction nodes and $n_1 n_2 \notin \mathcal{E}$. We define a path from n_1 to n_2 as a sequence of road segment edges, one connected to another, denoted as $n_1 n_{i_1}$, $n_{i_1} n_{i_2}, \ldots, n_{i_{k-1}} n_{i_k}$, $n_{i_k} n_2$ $(k > 0)$. The length of a path h between n_1 and n_2, denoted by $pathlength(h)$, is computed as $seglength(n_1 n_{i_1}) + seglength(n_{i_k} n_2) + \sum_{\alpha=1}^{k-1} seglength(n_{i_\alpha} n_{i_{\alpha+1}})$. Given two road junctions n_1 and n_2, since there can be more than one path from n_1 to n_2, we use $PathSet(n_1, n_2)$ to denote the set of all paths from n_1 to n_2. We define a segment length-based shortest path from n_1 to n_2, denoted by $sl\_shortestpath(n_1, n_2)$, as $\{h_{sl} | pathlength(h_{sl}) = \min_{h \in PathSet(n_1, n_2)} pathlength(h)\}$. The travel time of a road segment $n_i n_j$ is defined as $\frac{seglength(n_i n_j)}{speedlimit(n_i n_j)}$ and thus the travel time of a path h, denoted by

$traveltime(h)$, is calculated as $\frac{seglength(n_1 n_{i_1})}{speedlimit(n_1 n_{i_1})} + \frac{seglength(n_{i_k} n_2)}{speedlimit(n_{i_k} n_2)} + \sum_{\alpha=1}^{k-1} \frac{seglength(n_{i_\alpha} n_{i_{\alpha+1}})}{speedlimit(n_{i_\alpha} n_{i_{\alpha+1}})}$. The travel time-based shortest path from n_1 to n_2, denoted by $tt\_shortestpath(n_1, n_2)$, is defined as $\{h_{tt} | traveltime(h_{tt}) = \min_{h \in PathSet(n_1,n_2)} traveltime(h)\}$.

A *road network location*, denoted by $L = (n_i n_j, p)$, is a tuple of two elements: a road segment $n_i n_j$ and the *progress* p along the segment from n_i to n_j. The road network distance between two road network locations $L_1 = (n_{i_1} n_{i_2}, p_1)$ and $L_2 = (n_{j_1} n_{j_2}, p_2)$ is the length of the shortest path between L_1 and L_2 in terms of either segment length or travel time. The **segment length-based road network distance** and **travel time-based road network distance** are formally defined respectively as follows:

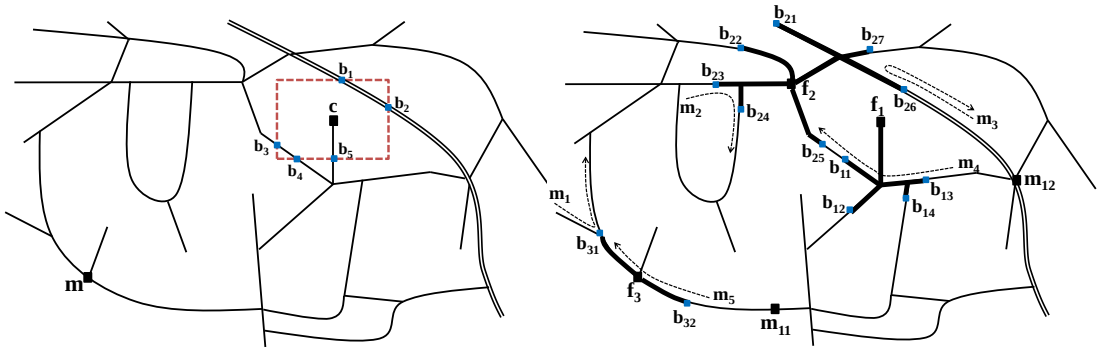
$sldistance(L_1, L_2) = seglength(n_{i_1} n_{i_2}) - p_1 + p_2 + pathlength(sl\_shortestpath(n_{i_2}, n_{j_1}))$

$ttdistance(L_1, L_2) = \frac{seglength(n_{i_1} n_{i_2}) - p_1}{speedlimit(n_{i_1} n_{i_2})} + \frac{p_2}{speedlimit(n_{j_1} n_{j_2})} + traveltime(tt\_shortestpath(n_{i_2}, n_{j_1}))$.

Even though the segment length-based distance is the most commonly used distance measure on road networks, it may not provide sufficient and accurate distance information in terms of actual travel time from the current location to the destination, considering that highway road segments are usually much longer but also with much higher speed limits and thus may have relatively shorter travel time compared to some local road segments. To ensure high accuracy and high performance of spatial alarm processing, we use the travel time-based distance as default road network distance measure in RoadAlarm.

5.2.2 Road Network-aware Spatial Alarms

In RoadAlarm, we define a road network-aware spatial alarm as a star-shaped subgraph centered at the alarm focal point, denoted as $SA_{RN}(p_f, r, S)$ where p_f is the alarm target or the alarm focal point (a road network location), r is the alarm monitoring region, represented by a spatial range (segment length or travel time) from p_f, and S is a set of subscribers. Consider Fig. 34(b) that shows three star-shaped alarms with focal points f_1, f_2, and f_3. The road network-aware spatial alarm with focal point f_1 has a range of 5 miles based on the segment length. The road network-aware spatial alarm with focal point f_2 has a range of 10 minutes based on the travel time. We call those points on the road network that



(a) Rectangular Spatial Alarm (b) Road Network Spatial Alarms

Figure 34: Spatial Alarms

bound a star-shaped spatial alarm *border points*. For example, b_{12} is one of the four border points of the alarm with focal point f_1.

However most existing techniques define spatial alarms using Euclidean distances and thus a rectangular region is typically used to represent the spatial alarm region of interest [28, 27, 85, 41]. A rectangular spatial alarm is defined as $SA_{Euc}(p_1, p_2, S)$ where p_1 and p_2 are the top-left and the bottom-right points respectively and S is a set of subscribers of this alarm. Fig. 34(a) shows an example rectangular alarm that has five intersecting points with the underlying road network. Such a spatial alarm can be triggered even though its subscribers' current location is far away from the alarm target c based on road network distance. For example, if a mobile subscriber of the alarm is located on b_1, the alarm should be triggered because the mobile subscriber is within the rectangular alarm region. However, even though b_1 is the nearest intersecting point to the alarm target c based on the Euclidean distance, its road network location is far away from the alarm target c based on the road network distance and thus the mobile subscriber can save its battery energy by sleeping for a longer time. This example illustrates the problem of using Euclidean distance to define spatial alarms and highlights the potential benefit of road network-aware spatial alarm processing.

5.2.3 Alarm Miss and Hibernation Time

We define an *alarm miss* as a case when a spatial alarm is not triggered as it should be even though a mobile subscriber of the alarm enters or passes through the alarm region.

Consider Fig. 34(b): moving objects m_1, m_2, m_3 and m_5 should each receive a spatial alarm alert and m_4 should get two alerts from the spatial alarms with alarm targets f_1 and f_2 sequentially. If any one of those alarms is not triggered during the course of travel for the five subscribers, the alarm miss has happened. Opposite to alarm miss, an *alarm hit* refers to the case when a spatial alarm is triggered when one of its mobile subscribers enters the alarm region.

As mentioned earlier, in RoadAlarm we compute a hibernation time for each mobile subscriber, during which the mobile subscriber hibernates the RoadAlarm thin client on her device. We define a *hibernation time* for each moving object, which is a time interval during which the moving object does not need to wake up and the alarm server does not need to perform alarm checks for this mobile subscriber. A hibernation time of a moving object is specified by a time interval consisting of its start time and end time. If the current time is between them, the moving object's current status is *hibernation*; otherwise, it is *alive*. Upon expiration of the current hibernation time, the mobile client wakes up, and communicates with the spatial alarm server to obtain a new hibernation time. The alarm server examines the current location of the mobile subscriber and the set of alarms subscribed by this subscriber to determine the new hibernation time. If the new hibernation time is smaller than a system-defined threshold δ, a spatial alarm is triggered and the mobile subscriber is notified. Otherwise, a new hibernation time will be sent to the mobile subscriber.

We would like to note that the timeliness of alarm triggering is also important, especially when spatial alarms are defined with some quality of service (QoS) guarantee. For example, it is possible that based on the current hibernation time for the moving object m_5 in Fig. 34(b), m_5 may receive the spatial alarm alert when it approaches the focal point f_3 or just before leaving the spatial alarm through the border point b_{31}. Thus, in RoadAlarm we use a stronger definition of alarm miss: if a moving object's current status is *hibernation* when it enters alarm region of a spatial alarm by crossing a border point of the alarm, we treat it as an alarm miss.

The *alarm success rate* is the percentage of spatial alarm alerts that are not missed, and

is defined as follows:

$$alarm\ success\ rate = 1 - \frac{Total\ number\ of\ alarm\ misses}{Total\ number\ of\ actual\ alarm\ hits}.$$

For example, if there are 9 alarm hits and 1 alarm miss (actual hit but not triggered), the success rate is 90%. Other metrics we use to evaluate the efficiency and effectiveness of road network-aware spatial alarm processing include the average hibernation time, the number of wakeups, the number of border points used in the computation of hibernation time, and the number of alarm checks upon each wakeup.

5.3 Spatial Alarm Processing

In this section we present the design consideration of the RoadAlarm baseline algorithm for efficient processing of spatial alarms. We first describe the Euclidean distance-based approach and the conventional network expansion-based approach to process road network-aware spatial alarms and analyze the problems with these two approaches. Then we introduce the baseline algorithm for RoadAlarm by incorporating subscription filter and Euclidean lower bound filter.

5.3.1 Euclidean Distance-based Approach

The Euclidean distance-based approach is often considered as the most intuitive baseline approach to implementing spatial alarm processing. Concretely, for every mobile object m, upon the expiration of its hibernation time, m wakes up and contacts the spatial alarm server to obtain its new hibernation time. The alarm server first retrieves the index of all spatial alarms and obtains the set of border points for each active spatial alarm. Then the alarm server computes the Euclidean distance between the current location of m and each of the border points for all spatial alarms and selects the border point that is the nearest to the current location of m, denoted by $b_{nearest}$, and calculates the new hibernation time for m based on the Euclidean distance and a velocity metric that is representative, such as the global maximum speed (V_{max}) or the expected speed of m ($V_{expected}$). For example, if there are 35 mph, 55 mph, and 65 mph road segments on the road network, the global maximum speed is 65 mph. Although using the global maximum speed is too

conservative to calculate the hibernation time, it ensures high alarm success rate. The end time of the new hibernation time for m based on this Euclidean distance-based method using the global maximum speed is defined as $\textit{current time} + \frac{eucdistance(m,b_{nearest})}{V_{max}}$ where $eucdistance(m, b_{nearest})$ is the Euclidean distance between m and $b_{nearest}$. The object m will be in the hibernation status during the above hibernation time interval.

In general, the Euclidean distance-based method using the global maximum speed is the most conservative technique since not all mobile objects are traveling at the maximum speed. Thus an alternative metric we adopted in the first prototype of RoadAlarm is the expected speed calculated using the current location of m, the previous location, the previous expected speed, and the previous maximum speed [85]. The end time of the new hibernation time for m based on this Euclidean distance-based method using the expected speed is defined as $\textit{current time} + \frac{eucdistance(m,b_{nearest})}{V_{expected}(m)}$ where $V_{expected}(m)$ is the expected speed of m.

Although the Euclidean distance-based approach is simple to implement, it suffers from a number of fatal weaknesses. First, the hibernation time is computed using the Euclidean distance rather than road network distance, thus the hibernation time is unnecessarily short. Consequently, mobile objects need to wake up frequently, making RoadAlarm consuming higher battery energy than necessary. Second, for each mobile object m, the nearest spatial alarm may not be subscribed by m, thus the hibernation time computed using the Euclidean distance to the nearest spatial alarm is misleading. This is especially true when all the spatial alarms subscribed by m is far away from the current location of m.

5.3.2 Network Expansion-based Approach

Another intuitive baseline approach to evaluating road network-aware spatial alarms is to use Dijkstra's network expansion algorithm [40]. We present two methods using different road network distances: one is using the segment length-based distance (*NE-S*) and the other is using the travel time-based distance (*NE-T*).

When a moving object m wakes up, *NE-S* and *NE-T* first retrieve a set of spatial

alarms (A_m) subscribed by m. For each spatial alarm $a_i \in A_m$, the set of border points of a_i are obtained. *NE-S* and *NE-T* take the current location of m and each border point of a_i as input to calculate the segment length-based shortest path and the travel time-based shortest path respectively, using Dijkstra's network expansion algorithm. After computing the shortest path from m's current location to every border point of all alarms in A_m, *NE-S* selects the shortest path with the smallest segment length-based distance, denoted by p_{sl}, to compute the new hibernation time for m. Similarly, *NE-T* chooses the shortest path having the smallest travel time-based distance, denoted by p_{tt}, to compute the new hibernation time for m. Thus, we can compute the end time of the hibernation time for m based on *NE-S* and *NE-T* as follows:

$$HT_{NE-S}(m) = \ current\ time + traveltime(p_{sl})$$

$$HT_{NE-T}(m) = \ current\ time + traveltime(p_{tt}).$$

Recall that $traveltime(p)$ computes the travel time of a path p as described in Section 5.2.1.

The network expansion-based approach is simple and straightforward to implement. However, the computation cost of this approach is extremely high since it examines all border points of every spatial alarm subscribed by a mobile object m at each wakeup. The shortest path computation cost to calculate the hibernation time of m increases rapidly as the number of alarms subscribed by the mobile object m increases and most of the subscribed alarms are far away from the current location of m. This is because the computation cost of Dijkstra's network expansion algorithm primarily depends on the size of underlying road network, the distance between the source location and the destination location, and the number of shortest path computations to be performed. If the destination is far away from the source, it is highly costly to compute the shortest path using the Dijkstra's network expansion algorithm because it exhaustively expands too many unnecessary nodes and edges.

5.3.3 RoadAlarm Baseline Approach

Bearing in mind the problems with the Euclidean distance-based approach and network expansion-based approach, we design the baseline approach of RoadAlarm by introducing

two simple and yet effective filters. We use the subscription filter to scope the computation of the hibernation time for each mobile object to only those alarms that are subscribed by the mobile object. In addition, we use *Euclidean lower bound (ELB)* as another type of filter to minimize the number of shortest path computations required to compute the hibernation time upon each wakeup by filtering out some irrelevant border points of subscribed spatial alarms. The concept of *Euclidean lower bound* refers to the fact that the segment length-based shortest path distance between two network locations L_1 and L_2 is at least equal to or longer than the Euclidean distance between L_1 and L_2. By combining the subscription filter and ELB filter, the RoadAlarm baseline approach (BA) can minimize the number of shortest path computations required for computing hibernation time for each mobile subscriber while maintaining the accuracy of alarm evaluation. We present two methods using the segment length-based and the travel time-based road network distance, denoted by *BA-S* and *BA-T* respectively.

Concretely, instead of computing shortest paths from the current location L_m of the mobile subscriber m to every border point of all alarms subscribed by m, *BA-S* computes the new hibernation time of m in five steps. *First*, for every alarm subscribed by m, denoted by $a_i \in A_m$, we find the border point that has the shortest distance from L_m. Instead of computing shortest paths from L_m to every border point of alarm a_i, we compute the Euclidean distance between L_m and every border point of a_i and sort the set of border points based on their Euclidean distances from L_m in an ascending order using the Incremental Euclidean Restriction (IER) algorithm [53, 103, 90]. *Second*, let b_{nn} denote the border point that has the smallest Euclidean distance from L_m. We compute the shortest path from L_m to b_{nn} using the segment length-based distance. *Third*, we use a binary search algorithm to examine the sorted list of border points and remove those border points whose Euclidean distance from L_m is bigger than *sldistance*(L_m,b_{nn}). *Fourth*, for each remaining border point b_j, *BA-S* computes the shortest path from L_m to b_j. If *sldistance*(L_m,b_j) $<$ *sldistance*(L_m,b_{nn}) holds, we assign b_j to be b_{nn}. Thus, for a given mobile object and an alarm $a_i \in A_m$, the nearest border point b_{nn} of a_i will be used as the reference border point of a_i to compute the hibernation time for m. *Finally*, *BA-S* examines every alarm $a_i \in A_m$

and its nearest border point b_{nn} and chooses the border point whose segment length-based distance from L_m is the smallest. Let b_{min} denote this nearest border point and p_{min} denote the shortest path from L_m to b_{min}. Thus we compute the end time of the new hibernation time for m as $current\ time + traveltime(p_{min})$.

Now we illustrate the working of the baseline approach using the example in Fig. 34(b). We have three spatial alarms a_1, a_2, a_3 with focal points f_1, f_2, f_3 respectively and two moving objects m_{11} and m_{12}. Let us assume that m_{11} subscribes to a_1 and a_3 and m_{12} subscribes to a_1 and a_2. Let $L_{m_{11}}$ and $L_{m_{12}}$ denote the current location of m_{11} and m_{12} respectively. When m_{11} and m_{12} wake up upon the expiration of their hibernation time, without the subscription filter and ELB filter, we will need to compute the shortest paths from $L_{m_{11}}$ and $L_{m_{12}}$ to all 13 border points and then choose the nearest border point, which has the shortest network distance (either segment length-based or travel time-based) from $L_{m_{11}}$ and $L_{m_{12}}$ respectively. With the subscription filter, we can filter out alarm a_2 for m_{11} and alarm a_3 for m_{12} when computing the new hibernation time. By the ELB filter, to find the new hibernation time for m_{12}, we only need to perform one shortest path computation from $L_{m_{12}}$ to b_{13}. This is because by Euclidean distance, b_{13} is the nearest border point of a_1 from $L_{m_{12}}$ and b_{26} is the nearest border point of a_2 from $L_{m_{12}}$. Given that $eucdistance(L_{m_{12}},b_{13}) < eucdistance(L_{m_{12}},b_{26})$, b_{13} is the nearest border point for m_{12}. Now we compute the network distance (either segment length-based or travel time-based) from $L_{m_{12}}$ to b_{13}, denoted by $sldistance(L_{m_{12}},b_{13})$. By comparing $sldistance(L_{m_{12}},b_{13})$ with the Euclidean distance from $L_{m_{12}}$ to all other border points of a_1 and a_2, we find that the following condition $eucdistance(L_{m_{12}},b_k) > sldistance(L_{m_{12}},b_{13})$ holds ($k = 11, 12, 14, 21, 22, 23, 24, 25, 26, 27$). Thus the ELB filter effectively removes the unnecessary shortest path computations when computing the new hibernation time for m_{12}.

We below show that the network location of the mobile object has significant impact on the effectiveness of the ELB filter. Consider the mobile object m_{11} and the two alarms a_1 and a_3 subscribed by m_{11} in Fig. 34(b). For the alarm a_3, we do not need to compute the shortest path from $L_{m_{11}}$ to the border point b_{31} since the Euclidean distance between $L_{m_{11}}$ and b_{31} is longer than the segment length-based distance from $L_{m_{11}}$ to b_{32}. However, for

the alarm a_1, the list of border points sorted in ascending order of their Euclidean distance from L_{m_1} is $\{b_{12}, b_{11}, b_{14}, b_{13}\}$. Clearly, the segment length-based network distance from $L_{m_{11}}$ to its nearest border point b_{12}, denoted by *sldistance*($L_{m_{11}}$, b_{12}), is longer than the Euclidean distance between $L_{m_{11}}$ and each of the last three border points in the list and thus none of the three border points are filtered out for alarm a_1.

BA-T finds the nearest border point of a moving object m using the travel time-based road network distance. For *BA-T*, we cannot directly use the ELB filtering as done in *BA-S* since the Euclidean lower bound property does not hold for the travel time-based distance. For example, when the Euclidean distance and the segment length-based distance between a border point and the current location of a mobile object m are 5 miles and 10 miles respectively, there could be another border point in which the Euclidean distance and the segment length-based distance are 12 miles and 15 miles respectively, but it has shorter travel time-based distance since there is a freeway connecting the border point and the current location of m. Therefore, we extend the ELB filtering for the travel time-based distance. Instead of using only segment lengths, *BA-T* defines the *travel time-based Euclidean lower bound* as the travel time multiplied by the global maximum speed limit on the entire road network. For example, if the travel time from the current location of m to an alarm border point is 1 hour and the global maximum speed limit is 70 mph, the travel time-based ELB in *BA-T* is 70 miles (1h x 70mph). *BA-T* excludes border points whose Euclidean distance is longer than 70 miles since the moving object m cannot get to those border points within 1 hour even if it travels at the global maximum speed. Since *BA-T* is using the global maximum speed limit to calculate the travel time-based ELB, the search space of *BA-T* is usually larger than that of *BA-S*, i.e., *BA-T* considers more border points of alarms subscribed by m to find the nearest one. The remaining steps of *BA-T* are the same as those in *BA-S*. In the first prototype of RoadAlarm we use the global maximum speed limit for travel time-based ELB in order to ensure the high accuracy of alarm evaluation. It would be interesting to use some less conservative speed or motion metrics to see if we can further reduce the search space needed for computing the hibernation time for mobile objects at the cost of a small and affordable accuracy loss.

5.4 Motion-aware Optimizations

Compared to the Euclidean distance-based approach and the conventional network expansion-based approach, the RoadAlarm baseline approach (BA) improves the efficiency of road network-aware alarm processing along two dimensions. First, it uses the subscription filter to narrow down the set of spatial alarms to be considered for computing the hibernation time upon wakeup of each mobile subscriber. Second, it utilizes the ELB filter to cut down the number of border points to be examined for shortest path computation while achieving high alarm success rate.

However, the ELB filter is not always effective. In some cases, the number of border points after applying the ELB filter remains to be high. Recall the case of m_{11} in Fig. 34(b) in which the ELB filter can filter out one border point (b_{31}) for alarm a_3. For alarm a_1, the Euclidean distance from $L_{m_{11}}$ to b_{12} is the shortest and thus the road network-based distance from $L_{m_{11}}$ to b_{12} is first calculated. Because this road network distance is longer than the Euclidean distances from $L_{m_{11}}$ to all the other border points (b_{11}, b_{13}, b_{14}), the ELB filter filters out none of border points for alarm a_1.

In this section we introduce a suite of motion-aware filters to further reduce the search space and the computation time of the RoadAlarm baseline approach (BA), especially for those moving objects that subscribe many spatial alarms and their alarms are scattered in a large geographical area. The main idea of the motion-aware filters comes from the observation that mobile objects traveling on road networks typically exhibit some degree of steady motion. First, a moving object traveling on a spatially constrained road network can move only by following the predefined road segments connected to the current road segment it resides. Thus, its movement cannot be changed drastically. For example, if a moving object is marching on a road segment, its current moving direction cannot be changed until it reaches a road junction. Furthermore, even if it reaches a road junction, it has high probability to follow the road segment in the same or similar direction at the junction node. We refer to such motion behavior as steady motion.

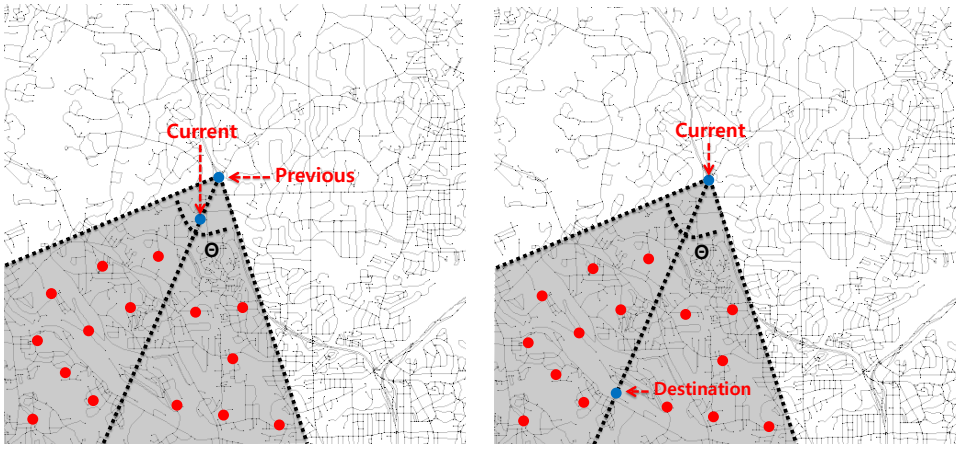
In this section, we present five types of steady motion-based filters. Our first three optimization techniques use steady motion degree Θ to capture the constrained motion

characteristics of moving objects traveling on a road network. For each mobile object, its steady motion degree Θ models the direction of its movements along the road network. If a sharp turn occurs at a junction node, a new Θ value will be computed for the mobile object based on the characteristics of underlying road networks and past movement history of the mobile object. We can also view this Θ as a confidence indicator. When a mobile object moves on the road network by following its current direction, a large Θ value indicates possible sharp turns and sudden travel direction changes whereas a small Θ value indicates high probability of steady motion along the current direction. When a mobile object is traveling on the road network with a clear destination in mind, this Θ angle can be determined based on the current location of the mobile object and the destination location.

5.4.1 Current Direction-based Motion-aware Filter

The first motion-aware filter is based on the current direction of moving objects and their steady motion degree Θ. This filter selects only those border points that reside in the Θ region anchored at the current location of the mobile object. The Θ degree is determined based on the current travel direction of the mobile object. One popular way to define the current direction of a moving object is to use the *current direction vector* in which we use the last reported location as the initial point and the current location as the terminal point of the vector. Let (p_1, p_2) and (c_1, c_2) denote the previous location and the current location of a moving object m respectively. The current direction vector of m is defined as $v = <c_1 - p_1, c_2 - p_2>$. Based on this current direction vector, when a mobile object m wakes up, this filter limits the search space using the steady motion degree Θ and selects only those border points of m that reside within this reduced search space, as shown in Fig. 35(a). For example, let (xb_1, xb_2) denote a border point. To check if the border point is within the Θ reduced search space, this filter first defines another vector $w = <xb_1 - p_1, xb_2 - p_2>$ and then calculates the degree of the border point from the current direction vector v using the following equation:

$$sm\_degree(v, w) = \arccos(\frac{v \cdot w}{|v||w|}).$$



(a) Current direction-based (b) Destination-based

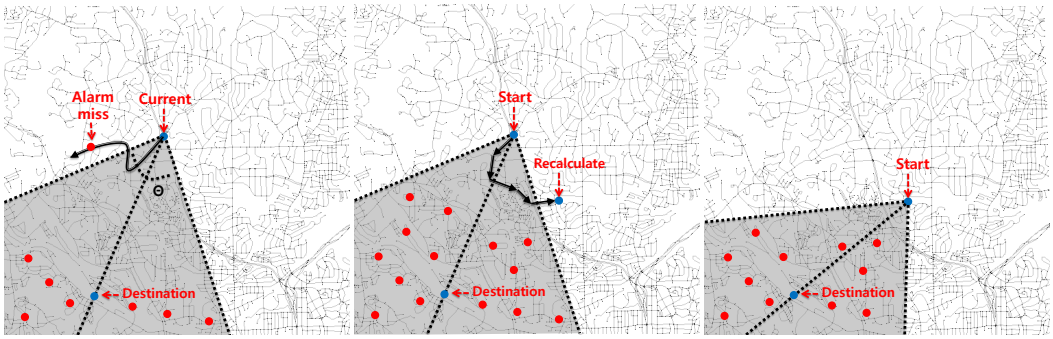
Figure 35: Vector-based Motion-aware Filters

If $sm\_degree(v, w) > \Theta$, this border point is removed since it is outside the constrained search space. For the remaining unfiltered border points, our approach calculates the new hibernation time by executing the RoadAlarm baseline approach.

The current direction-based motion-aware filter is good when the hibernation time is relatively short and the time window in which the mobile object moves steadily is relatively low as well, since the current direction vector changes each time when the mobile object wakes up due to the expiration of its current hibernation time. Furthermore, the current direction-based motion-aware filter may be suitable for some mobile clients who do not want to disclose their destination information due to privacy reasons. However, if the destination is given (or can be inferred by using its calendar application), it is more effective to use a destination-based motion-aware filter.

5.4.2 Destination-based Motion-aware Filter

The destination-based motion-aware filter utilizes both the current location and the destination information of moving objects. Destination information can be directly given by the mobile clients, such as those using car navigational systems or can be extracted from mobile clients' calendar applications. In this filter, the degree Θ indicates how confident the moving object will march toward its destination. The destination-based motion-aware filter chooses only border points that reside in the Θ region defined based on the current



(a) Alarm miss (b) Cache Invalidation (c) New search space

Figure 36: Caching-based Motion-aware Filter

location of moving objects to their destination. We define a *destination vector* to represent the direction toward the destination, in which the current location and the destination location are used as the initial and terminal point of the vector respectively. Let (c_1, c_2) and (d_1, d_2) denote the current location and the destination of a moving object m respectively. The destination vector of m is defined as $v = <d_1 - c_1, d_2 - c_2>$. When m wakes up, the destination-based motion-aware filter restricts the search space using the destination vector v and Θ and then selects only the border points within this Θ restricted search space, as shown in Fig. 35(b). For example, let (yb_1, yb_2) denote a border point. To check if the border point is within the reduced search space, this filter first computes another vector $w = <yb_1 - c_1, yb_2 - c_2>$ and then calculates the degree of this border point in terms of this new vector and the destination vector v using the equation $sm\_degree(v, w)$. We remove those border points whose $sm\_degree(v, w)$ values are higher than the specific Θ defined by m or calculated by the system in the absence of user-defined Θ. Our approach calculates the nearest border point by examining all the unfiltered border points and then computes the new hibernation time for m by invoking our RoadAlarm baseline algorithm.

5.4.3 Caching-based Motion-aware Filter

Both the current direction-based motion-aware filter and the destination-based motion-aware filter can reduce the computation cost of finding the nearest border point for each mobile object upon its wakeup. This computation reduction is achieved by reducing the number of candidate border points and thus the search space through a combination of the

steady motion degree Θ with current direction or destination information. However, those two filters also suffer from a couple of inherent problems. First, if a mobile object m takes a short detour due to traffic and moves slightly out of the scoped spatial region defined by Θ and the current direction or destination, there could be an alarm miss. Consider Fig. 36(a): a moving object m has moved slightly outside the scoped spatial region and there exists a spatial alarm that resides just outside the scoped region and is very close to the current location of m. Unfortunately, this alarm will be missed if we use the current direction-based motion-aware filter or destination-based motion-aware filter. Another weak point is that those two filters recalculate the search space and thus the set of candidate border points at every wakeup of each moving object. This causes not only unnecessarily frequent and possibly duplicate computation of the candidate border points but also adds some unnecessary susceptibility to small detour-like movements of mobile objects. Concretely, if a moving object changes its direction slightly, for example, due to traffic directed detour, the selection of the candidate border points found at the current wakeup could be very different from the selection at the previous wakeup. Therefore, the two filters may miss some spatial alarms that have high probability to be a hit due to this unnecessarily sensitive susceptibility.

To address this problem, we propose another motion-aware filter, called *caching-based motion-aware filter*, based on the observation that moving objects will move toward their destination constantly and persistently even though they may change their direction opposite to (or deviate quite bit from) the destination for a short period of time. Initially, this filter selects the candidate border points for each moving object based on its current location, destination location, and steady motion degree Θ using the destination-based motion-aware filter. This filter then stores the selected candidate border points with the calculated destination vector for each moving object. When a moving object m wakes up next time, instead of recomputing the Θ region and the set of candidate border points as done in the destination-based motion-aware filter, this caching-based motion-aware filter retrieves the stored candidate border points of m and then finds the nearest border point to the current location of m by examining the stored border points using our RoadAlarm

baseline approach. Finally, this approach calculates the hibernation time using the nearest border point in the same way as is done in the baseline approach.

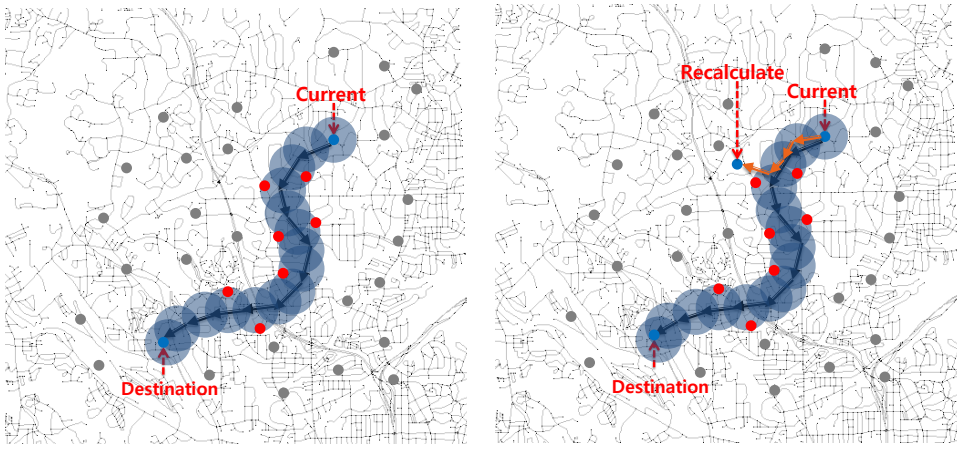
Even though the caching-based motion-aware filter is proposed to handle the susceptibility to small changes, continuous small changes can make a big change as shown in Fig. 36(b). To address this problem, this filter has a mechanism to check whether the stored border points are obsolete and thus to calculate new candidate border points. When a moving object wakes up, this filter calculates the degree of the moving object's current location from the stored destination vector. If the degree is larger than Θ (i.e., the object went out of the scoped search space), then this filter recalculates the search space based on the object's current location as shown in Fig. 36(c).

5.4.4 Shortest Path-based Motion-aware Filter

Even though the caching-based motion-aware filter avoids unnecessarily frequent filtering of border points, it still needs to examine too many border points in order to find the nearest one, especially when Θ is large and many alarms are subscribed by moving objects. Consider Fig. 35(b): the border points on the bottom far left or far right corner are unlikely to be hit by the moving object since it is far away from the object's destination. Motivated by this observation, we propose the shortest path-based motion-aware filter based on a natural assumption that moving objects will follow the shortest path to the destination. Initially, this filter calculates the shortest path (p_{min}) from the current location to the destination for each moving object and then selects some border points within a boundary distance d from the shortest path, as shown in Fig. 37(a). The distance d indicates the level of steadiness. For example, if a moving object follows the calculated shortest path, a small value of d is sufficient. To check if a border point b is within the boundary distance d from the shortest path p_{min}, this filter calculates the perpendicular distance from the border point to all road segments of p_{min} and then finds the minimum value as follows:

$$min_{b,p_{min}} = min_{n_{p_i}n_{p_{i+1}} \in p_{min}} pdistance(b, n_{p_i}n_{p_{i+1}})$$

where $n_{p_i}n_{p_{i+1}}$ is a constituent road segment of the path p_{min} and $pdistance(b, n_{p_i}n_{p_{i+1}})$ is the perpendicular distance from border point b to road segment $n_{p_i}n_{p_{i+1}}$. If $min_{b,p_{min}}$ is less



(a) Selection of border points (b) Shortest path recalculation

Figure 37: Shortest Path-based Motion-aware Filter

than d, the border point is selected as a candidate border point of the moving object. The shortest path-based motion-aware filter then stores the selected candidate border points with the calculated shortest path for each moving object. When a moving object m wakes up, this filter retrieves the stored candidate border points of m and then finds the nearest border point, among the retrieved border points, using the RoadAlarm baseline algorithm. Finally, this approach calculates the hibernation time using the nearest border point in the same way as is done in the baseline approach.

Like the caching-based motion-aware filter, the shortest path-based motion-aware filter also has a mechanism to handle moving objects that go out of our reduced search space based on the shortest path as show in Fig. 37(b). When a moving object wakes up, this filter calculates the distance from the stored shortest path of the object and, if the distance is larger than d, recalculates the search space based on the object's current shortest path to the destination.

5.4.5 Selective Expansion-based Motion-aware Filter

The shortest path-based motion-aware filter selects border points that have high probability to be hit based on the shortest path from the current location of moving objects to their destination. Even though it reduces the computation time to calculate the hibernation time by considering fewer (but more relevant) border points compared to the RoadAlarm

baseline approach and the other steady motion-based approaches, it still needs at least two shortest path computations: one for calculating the shortest path from the current location to the destination to select candidate border points and the other for choosing the nearest border point among the selected border points. These computations will increase the server loads when the destination is far away from the current location of a moving object and there is no nearby spatial alarm from the current location. To reduce the computation cost, we propose a selective expansion-based motion-aware filter in which an exact shortest path computation is not needed. The selective expansion-based filter expands only road segments that have high probability to be passed by a moving object. To select target road segments to be expanded, we utilize the destination of moving objects and apply the concept of Simulated Annealing (SA) to the expansion. SA probabilistically finds a good approximation to the global optimal solution in a large solution space by giving high randomness in early stages and almost no randomness in ending stages. Using this basic concept of SA, the selective expansion-based filter expands most of road segments in early steps even though some of them have opposite direction to the destination. In following steps, this filter incrementally strengthens the condition of the expansion and thus only road segments whose direction points toward the destination are expanded. This expansion is terminated if it satisfies one of three cases: 1) the expansion arrives at the destination, 2) the expansion meets any spatial alarm of the moving object, and 3) there is no more road segment that satisfies the condition of the expansion. Since this filter expands only relevant road segments that have high probability to be hit from the current location to the destination and it terminates the expansion process even though there is no found border point (case 3), it considerably reduces the computation cost to calculate the hibernation time compared to other processing methods, which require shortest path computations. In addition to the reduced computation cost, since it expands most of road segments in early steps, the selective expansion-based filter can cover common cases in which moving objects move in the opposite direction from the destination to take faster roads such as freeways.

The algorithm of the selective expansion-based filter is formally defined as follows. Like other processing methods we propose, this filter starts when a moving object m wakes up.

Let L_m and d_m denote the current location and the current destination of m respectively. We define $T(i)$, which denotes a time-varying parameter at step i and $E(n_p, i)$, which denotes an energy of an expansion node on a road junction n_p at step i. A smaller energy of a road junction means that the road junction has higher probability to be visited by m compared to other road junctions having a larger energy. A road segment $n_p n_q$ connected to n_p is expanded if a new energy $E(n_q, i+1)$, defined as follows, is less than $T(i)$.

$$E(n_q, i+1) = E(n_p, i) \times dv(n_p n_q, d_m)$$

where $dv(n_p n_q, d_m)$ represents a deviability of $n_p n_q$ from the destination d_m. Road segments whose direction points toward the destination have a small dv value and, on the other hand, road segments in which their direction is opposite to the destination have a large dv value. Therefore, road segments in which their direction points toward the destination will have higher probability to be expanded since their dv value is small. The deviability is defined as follows:

$$dv(n_p n_q, d_m) = \frac{degree(\overrightarrow{n_p n_q}, \overrightarrow{n_p d_m})}{180^\circ} \times \frac{1}{w(speedlimit(n_p n_q))}$$

where $w(speedlimit(n_p n_q))$ is a weight based on the speed limit of $n_p n_q$. By giving more weights to faster roads such as freeways, it makes such faster roads have higher probability to be expanded than slower roads. If $degree(\overrightarrow{n_p n_q}, \overrightarrow{n_p d_m})$ is 0° (i.e., $n_p n_q$ exactly points toward the destination), we use 1° instead of 0° to continue the selective expansion.

T value gradually decreases as steps increase to strengthen the expansion condition and is defined as follows:

$$T(i) = \frac{T_0}{k^i}$$

where k is a parameter which controls the expansion rate and T_0 is an initial value for the expansion. For a large k, the T value becomes smaller rapidly as steps increase and thus more road segments are excluded from the expansion, compared to a small k. A large T_0 value makes more road segments to be expanded. We use 1 as T_0 value to ensure that all road segments are expanded regardless of the k value and their dv value at first step.

The selective expansion-based filter starts with expanding the road junction n_0, where the current location L_m of m is located, with its initial energy 1 (i.e., $E(n_0, 0) = 1$). If L_m is

located on the road segment $n_p n_q$, this filter treats L_m as a road junction n_0 and $n_0 n_p$ and $n_0 n_q$ as road segments connected to n_0. For each road segment $n_0 n_j$ connected to n_0, this filter calculates $E(n_j, 1)$ using the above formula and then expands $n_0 n_j$ if $E(n_j, 1)$ is less than $T(0)$. While expanding $n_0 n_j$, this filter stops the whole expansion process if there is a border point of m or the destination d_m on $n_0 n_j$. If $n_0 n_j$ is expanded without encountering any border point or d_m, n_j is inserted into the expansion list for the next step with its energy $E(n_j, 1)$. After checking all road segments connected to n_0, the selective expansion-based filter moves to the next step and expands road junctions in the expansion list using the above process. This expansion process is terminated if there is no road junction for the next step or any border point or d_m is encountered during the expansion.

To calculate the hibernation time for m, if a border point or d_m is encountered during the expansion, this filter uses the travel time, from L_m to the encountered border point or d_m, as the hibernation time. Since this filter keeps the accumulated travel time from L_m to each expanded road junction, no additional computation is needed to calculate the hibernation time for m. If the expansion is terminated because there is no more road junction to be expanded, this filter chooses a terminal road junction (i.e., in which no connected road segment is expanded) having the smallest travel time among selected candidate terminal road junctions and then uses the travel time to the terminal road junction as the hibernation time for m. To select the candidate terminal road junctions, we introduce another confidence degree Θ_{SESM}. The selective expansion-based filter checks only terminal road junctions within Θ_{SESM} based on the vector from L_m to d_m and then chooses a terminal road junction having the smallest travel time among the candidate terminal road junctions. If Θ_{SESM} value is too large, it ensures high success rate, but its hibernation time is unnecessarily short because some terminal road junctions that are terminated at earlier steps and thus have short travel time are included in the search space. On the other hand, if Θ_{SESM} value is too small, it cannot ensure high success rate because only terminal road junctions that survived until last steps are included in the search space and thus the selected travel time is too long. Since this filter also keeps and updates the smallest travel time based on Θ_{SESM} during the expansion, no additional computation is needed to calculate the hibernation

time.

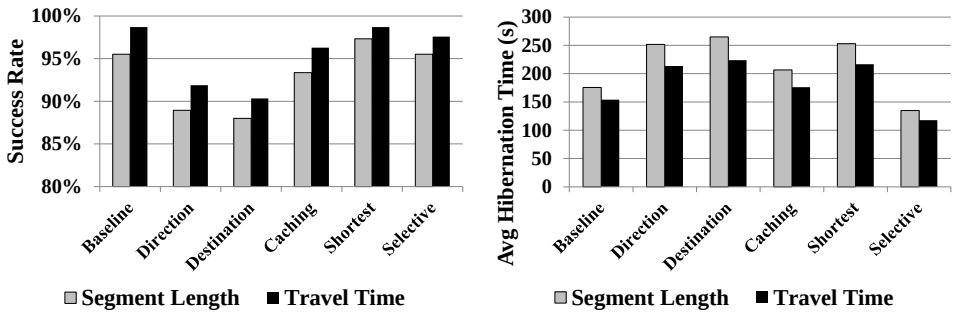
One disadvantage of the above synchronous (i.e., all target road junctions are expanded at the same step) selective expansion on road networks is that, even though a border point or the destination d_m is encountered during the expansion, the point could be reached by other road segments having shorter travel time at later steps. Furthermore, nearby border points could not be reached during the expansion if there are many short road segments from L_m to the border points. Therefore, spatial alarms can be missed due to the long travel time calculated by ignoring some nearby border points or faster road segments connecting to the border points. To solve this problem, we use an asynchronous version in which a road junction having the smallest segment length ($SESM-S$) or travel time ($SESM-T$) is expanded first, regardless of its current step, using a priority queue.

5.5 Experimental Evaluation

In this section we evaluate the performance of our RoadAlarm methods through four sets of experiments. We first compare our approaches with existing Euclidean space-based methods in terms of six measurements: alarm success rate, hibernation time, number of wakeups, total computation time, number of border points, and total alarm checking time. This set of experiments verifies that the shortest path-based motion-aware filter reduces the computation cost of servers and conserves energy of mobile clients while ensuring high success rate, and the selective expansion-based motion-aware filter reduces the computation cost of servers considerably while ensuring high success rate. The second set of experiments evaluates the effect of different steady motion degree Θ values. The third set of experiments measures the scalability of our approaches by varying the number of moving objects and the number of spatial alarms. The last set of experiments examines the effect of three types of road networks (urban, suburban, and rural) on the performance of the RoadAlarm approach.

5.5.1 Experiment Setup

We use *gt-mobisim* simulator [11] to generate mobility traces on real road networks downloaded from U.S. Geological Survey (USGS [22]). For the first three sets of experiments, the



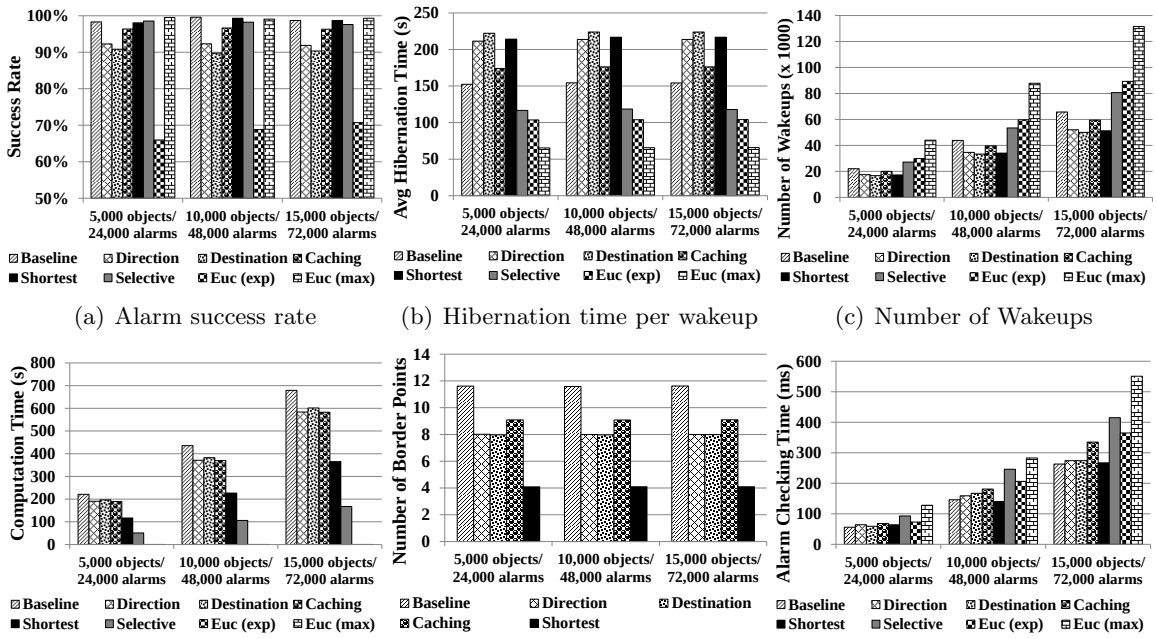
(a) Alarm success rate (b) Average Hibernation time

Figure 38: Segment Length-based vs Travel Time-based Approaches

mobility traces are generated on a map of northwest Atlanta, which covers about 11 km (6.8 miles) by 14 km (8.7 miles), using the random trip model [94]. The road networks consist of four different road types: residential roads and freeway interchange with 30 mph speed limit (48 km/h), highway with 55 mph limit (89 km/h), and freeway with 70 mph limit (113 km/h). Ranges of spatial alarms are chosen from a Gaussian distribution with a mean of 50 m and standard deviation of 10 m. We use 50 m as the boundary distance d of the shortest path-based motion-aware filter. For the selective expansion-based motion-aware filter, we empirically use 4 as the k value and 90° as the Θ_{SESM} value to select not too short and not too long travel time. We give 1, 2, and 3 to 30 mph, 55 mph, and 70 mph road segments respectively as their speed weights in order to give faster roads more chances of expansion.

5.5.2 Comparison with Existing Methods

We first compare segment length-based approaches and travel time-based approaches as shown in Fig. 38. These experiments use 15,000 objects (and about 72,000 spatial alarms) and 180° as the Θ value of the current direction-based, destination-based and caching-based motion-aware filters. Each object has different number of spatial alarms, given by Zipf distribution with five alarms as the most common value (i.e., rank 1). We exclude the results of network expansion-based methods since they cannot scale to 15,000 moving objects. The alarm success rate for travel time-based approaches is higher than the corresponding segment length-based approaches as shown in Fig. 38(a). This is primarily because segment length-based approaches select the segment length-based shortest path in which spatial alarms can



(a) Alarm success rate (b) Hibernation time per wakeup (c) Number of Wakeups

(d) Total Computation Time (e) Number of Border Points (f) Total Alarm Checking Time

Figure 39: Comparison with Euclidean Space-based Approaches

be missed if moving objects follow paths having shorter travel time. On the other hand, the average hibernation time of each travel time-based approach is shorter than that of its corresponding segment length-based approach since the travel time on the segment length-based shortest path is always equal to or longer than that on the travel time-based shortest path for the same source and destination location. Without loss of generality, in the rest of the experiments, we include the results of only travel time-based approaches for simplicity.

The first set of experiments compares our approaches with existing Euclidean space-based methods in Fig. 39. This set of experiments uses a moving object population with size ranging from 5,000 to 15,000 and each object has different number of spatial alarms, given by Zipf distribution with five alarms as the most common value.

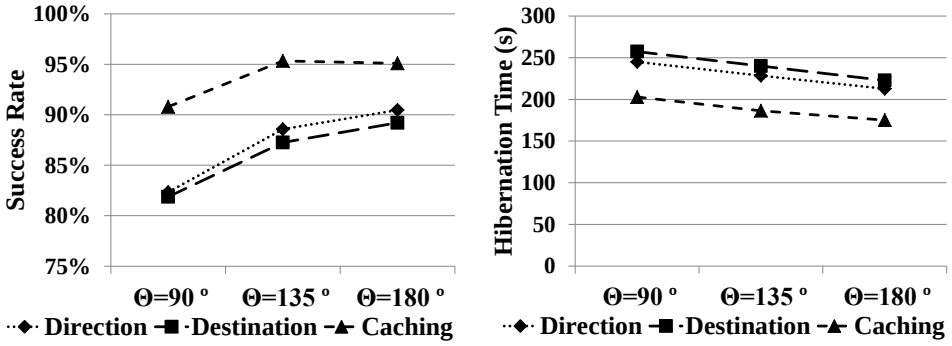
Alarm success rate. The success rates for different approaches are shown in Fig. 39(a). The shortest path-based and selective expansion-based motion-aware filters have almost the same success rate as the Euclidean distance-based approach using the global maximum speed and the RoadAlarm baseline approach. The caching-based filter has more than 5% better success rate than the destination-based filter. This confirms our assumption that moving objects will move toward their destination constantly even though they may change

their direction opposite to the destination for a short time. The Euclidean distance-based approach using the expected speed has the lowest success rate since it fails to consider spatial constraints of moving objects.

Hibernation time. Fig. 39(b) shows the average hibernation time of moving objects. The longer the hibernation time is, the more energy the mobile clients can conserve. The hibernation time of the shortest path-based filter is three times longer than that of the Euclidean distance-based approach using the global maximum speed and 40% longer than that of the RoadAlarm baseline approach. This result also shows that the shortest path-based filter ensures high success rate in the same way as the Euclidean distance-based approach and the RoadAlarm baseline approach even though moving objects of the shortest path-based filter can conserve much more energy. The selective expansion-based filter has 45% and 25% shorter hibernation time than the shortest path-based filter and the RoadAlarm baseline approach respectively since it calculates the hibernation time even though there is no found border point. It, however, still has 80% longer hibernation time than the Euclidean distance-based approach using the global maximum speed. The Euclidean distance-based approach using the global maximum speed has the shortest hibernation time since it utilizes the Euclidean distance and the global maximum speed to calculate the hibernation time.

The number of wakeups. Fig. 39(c) shows that the number of wakeups is inversely related to the hibernation time. The smaller number of wakeups indicates the lower server loads since the server computes the hibernation time whenever a moving object wakes up.

Computation time. Fig. 39(d) shows the total computation time to calculate the hibernation time. The shortest path-based filter has 45% faster computation time than the RoadAlarm baseline approach since it has smaller number of wakeups of moving objects. The selective expansion-based filter has the smallest computation time among the road network-based approaches since it does not need shortest path computations to calculate the hibernation time. Its computation time is just 24% and 45% of that of the RoadAlarm baseline approach and the shortest path-based filter respectively. The Euclidean distance-based approaches need only a little computation time since the computation cost to calculate the Euclidean distance is negligible, compared to the road network distance



(a) Alarm success rate (b) Hibernation time

Figure 40: Effects of the Steady Motion Degree Θ

calculation, even though the number of wakeups is more as shown in Fig. 39(c).

The number of border points. Even though there is only about 20% difference between the shortest path-based filter and the RoadAlarm baseline approach in terms of the number of wakeups, there is about 45% difference in terms of the computation time. Furthermore, even though all motion-aware filters have similar number of wakeups, only the shortest path-based filter has better computation cost than the others. This is because the shortest path-based filter considers the smallest number of border points to calculate the hibernation time, as shown in Fig. 39(e).

Alarm checking time. Fig. 39(f) shows the total processing time to check whether moving objects hit any alarms. We use a hash map to store spatial alarms. The result confirms that our approach checks spatial alarms efficiently.

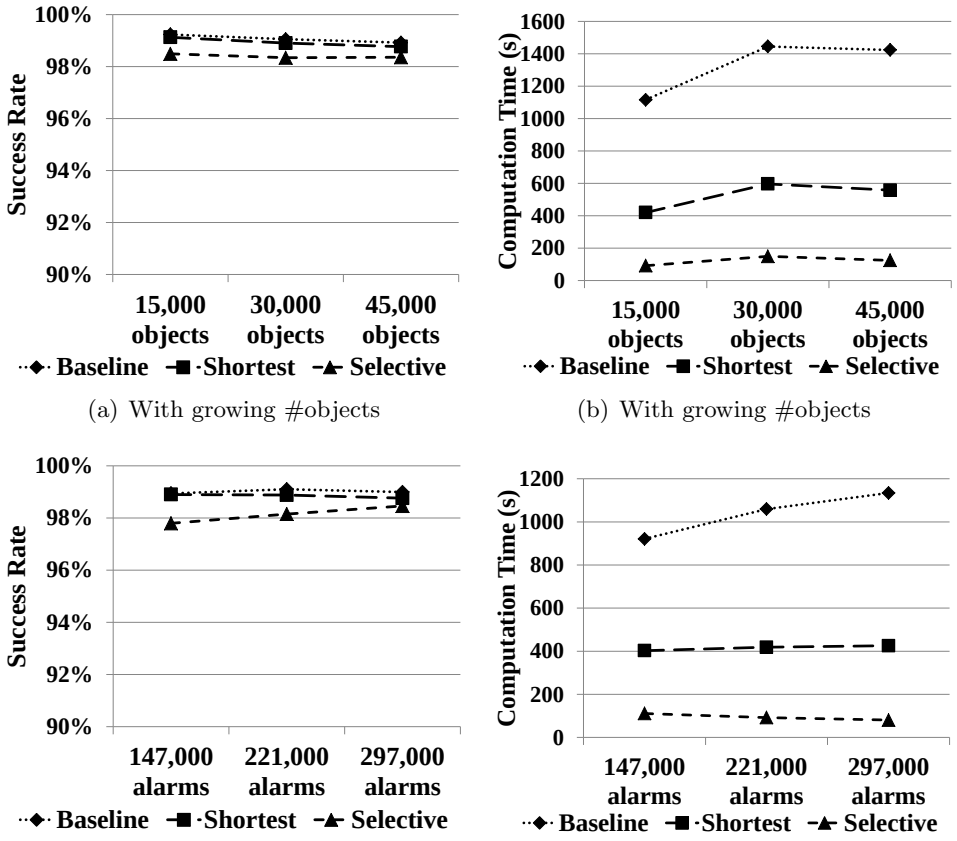
5.5.3 Effects of the Steady Motion Degree

We investigate the effect of different settings of the steady motion degree Θ on success rate and hibernation time with 15,000 moving objects and about 72,000 spatial alarms. The results are shown in Fig. 40 with Θ values set to 90°, 135° and 180°. The success rate for the current direction-based, destination-based and caching-based filters increases as Θ values increase, because more border points are selected as shown in Fig. 40(a). Fig. 40(b) shows that the average hibernation time decreases with growing Θ values. This is because border points having shorter travel time are newly selected to calculate the hibernation time as the search space increases.

5.5.4 Effects of the Growing Number of Objects and Alarms

Fig. 41(a) and Fig. 41(b) evaluate the scalability of our approaches by increasing the number of moving objects. Total 300,000 spatial alarms are deployed for this set of experiments and the number of moving objects increases from 15,000 to 45,000. Each object has zero to 30 spatial alarms, given by Zipf distribution with 15 alarms as the most common value, and all spatial alarms are private. We think this setting deploying 45,000 moving objects is realistic on this road network of northwest Atlanta, in which the total length of all road segments is 1384 km (865 miles), since there is a mobile user every 31 m (10 feet) on average. We include the measurement results of only the RoadAlarm baseline approach, the shortest path-based filter and the selective expansion-based filter as they have high success rate compared to other methods. Fig. 41(a) confirms that our approaches ensure the high success rate with growing number of moving objects. The selective expansion-based filter has slightly lower success rate than the others since it does not try to find the nearest border point if there is no nearby border point. In terms of the total computation time, there is no increase from 30,000 to 45,000 objects since with fixed alarms, many objects have no spatial alarms as shown in Fig. 41(b). The selective expansion-based filter's computation time is only 23% and 9% of that of the shortest path-based filter and the RoadAlarm baseline approach respectively while ensuring similar success rate.

Fig. 41(c) and Fig. 41(d) show the scalability of our approaches by increasing the number of spatial alarms with 15,000 moving objects. We increase the most common value of Zipf distribution from 10 to 20 and thus the total number of alarms grows from about 147,000 to 297,000. Fig. 41(c) verifies that our approaches ensure the high success rate with increasing number of spatial alarms. Note that the success rate of the selective expansion-based filter increases as the number of spatial alarms grows. This is because, if a moving object has more spatial alarms, there is a higher probability that a border point of the object is found during the selective expansion. Fig. 41(d) shows that the computation time of the shortest path-based filter increases only slightly with the growing number of spatial alarms. This is primarily because the shortest path-based filter selects only border points having high probability to be hit and thus the increased number of spatial alarms has no huge impact on



(a) With growing #objects

(b) With growing #objects

(c) With growing #alarms

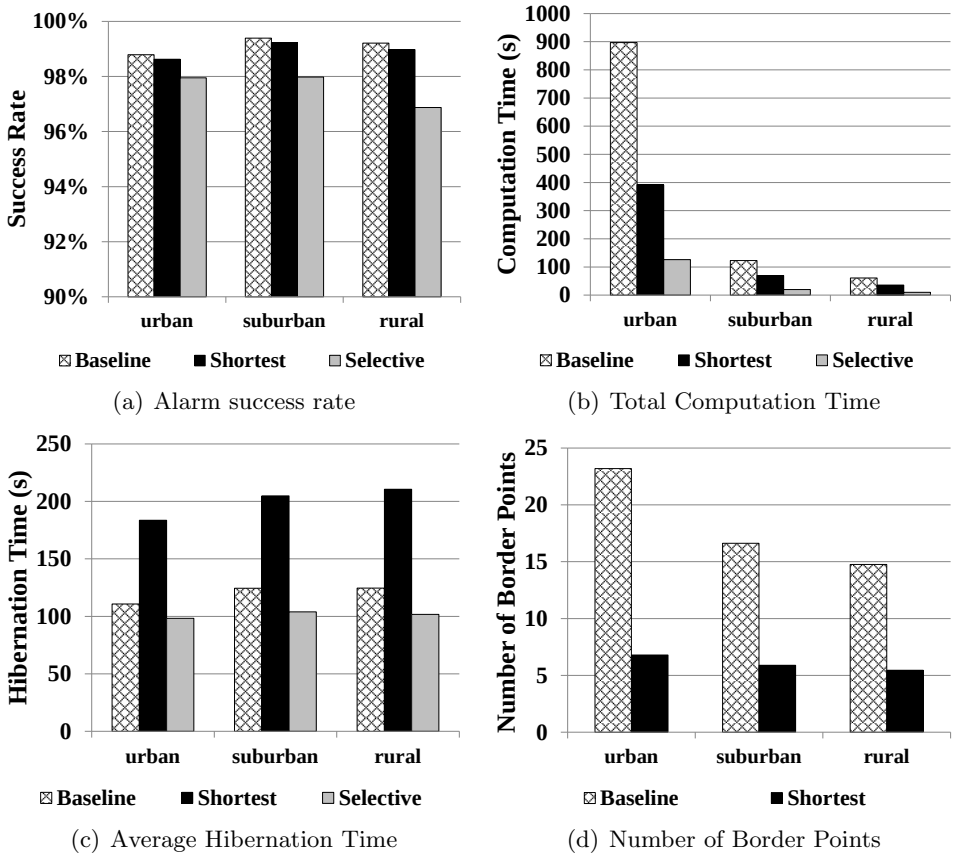
(d) With growing #alarms

Figure 41: Effects of the Growing Number of Objects and Alarms

the selected border points by the shortest path-based filter. Even though the computation time of the RoadAlarm baseline approach has more increase than that of the shortest path-based filter, it does not increase linearly with the growing number of spatial alarms. The computation time of the selective expansion-based filter even decreases as the number of alarms increases because the selective expansion of the filter is terminated earlier with the fewer number of expanded road segments due to the higher probability that a border point is found.

5.5.5 Effects of Different Road Networks

This set of experiments measures the performance of our approaches using different types of road networks. In addition to the map of northwest Atlanta as an urban road network, we choose the map of Duluth, GA and the map of Helen, GA as a suburban and a rural road network respectively. All three road networks cover almost same size (i.e., 6.8 miles



(a) Alarm success rate

(b) Total Computation Time

(c) Average Hibernation Time

(d) Number of Border Points

Figure 42: Effects of Different Road Networks

by 8.7 miles) but have totally different number of road segments and road junctions. The total numbers of road segments of the urban, suburban and rural road network are 9,187 (average length is 150.7 m), 1,600 (258.3 m) and 765 (356.3 m) respectively. The total numbers of road junctions are 6,979, 1,486 and 711 for the urban, suburban and rural road network respectively. The urban road network has 431 and 681 road segments having 70 mph and 55 mph speed limit respectively. The other road segments have 30 mph speed limit. 24 and 218 road segments of the suburban road network have 70 mph and 55 mph as their speed limit respectively. The rural road network has 27 and 66 road segments having 70 mph and 55 mph speed limit respectively.

Fig. 42 shows the experimental results for the three different road networks. This set of experiments uses 15,000 moving objects and total about 72,000 spatial alarms. Each object has different number of spatial alarms, given by Zipf distribution with five alarms as the most common value. Fig. 42(a) confirms that our approaches ensure the high success

rate for different types of road network. The selective expansion-based filter has slightly lower success rate on the rural road network since moving objects are more likely to use road segments whose direction does not point toward the destination and thus not expanded, due to the limited number of road segments. On the rural road network, the computation time is about 8% of that on the urban road network as shown in Fig. 42(b). This is primarily because of the high complexity (i.e., 12 times more road segments and 10 times more road junctions than the rural road network) of the urban road network. Fig. 42(c) shows that moving objects on the suburban and rural road networks have longer hibernation time than those on the urban road network even though the number of spatial alarms for each moving object and the focal point and the range of each spatial alarm are given by same distributions for all three road networks. Since the urban road network has 12 times more segments and 16 times more segments having 70 mph speed limit compared to the rural road network, it has more probability to have a path having shorter travel time between two locations. As shown in Fig. 42(d), fewer border points are considered to calculate the hibernation time on the suburban and rural road networks than on the urban road network because spatial alarms on complex road networks usually have more border points.

5.5.6 Summary

In summary, our experimental results show that the shortest path-based motion-aware filter outperforms the rest in most cases since this approach ensures high success rate while reducing the computation cost of servers and conserving energy of mobile clients. Since the selective expansion-based filter considerably reduces the computation cost while ensuring high success rate, it is suitable for spatial alarm processing servers that should compute the hibernation time quickly for a huge number of moving objects while ensuring longer hibernation time than the Euclidean space-based approach to save the energy of moving objects. For those applications in which high success rate is required, both the RoadAlarm baseline approach and the shortest path-based motion-aware filter are good options. Especially, for some applications in which the battery power of mobile clients is not a serious problem, the RoadAlarm baseline algorithm may be a better choice since it has a slightly higher

success rate than the shortest path-based motion-aware filter. The current direction-based filter, the destination-based filter and the caching-based filter are appropriate for those applications in which reducing the computation cost of servers and the battery consumption of mobile clients are top priorities while maintaining the acceptable success rate (about 90%).

5.6 Related Work

There are many existing studies on continuous spatial queries to find objects within a predefined range or k nearest objects from a query center point [107, 96, 111, 58, 54, 118]. Some of them are based on road networks [38, 84] or land surface [117]. However, spatial alarms are fundamentally different from continuous spatial queries in terms of their purposes as well as target applications. Continuous queries such as "find 3 nearest Starbucks stores while driving to Miami" require continuous query evaluation as I am driving on the US highway. On the other hand, spatial alarms have a predefined location of interest, such as "alert me when I am within 5 miles of the public library in Buckhead" and thus require alarm evaluation only when subscribers are in the vicinity of the spatial alarms. Even when the mobile subscribers are moving on the road, their spatial alarms may not need to be evaluated if those alarm targets are located far away from the current locations of their subscribers. This is the fundamental reason why spatial alarms deserve to be processed more efficiently using a different set of algorithms and optimizations.

Existing research on spatial alarms and location reminders mainly focuses on the Euclidean space. [28] proposes an approach to process spatial alarms in the Euclidean space by combining spatial indexes such as R-tree and Voronoi Diagram with the maximum speed-based safe period. [27] develops a safe region-based approach for spatial alarm processing in the Euclidean space. Different shapes of safe regions are proposed and compared in [27]. [41] points out the high cost of safe region-based approach and proposes the Mondrian tree index that can index both spatial alarms and alarm free regions within a uniformed framework. To the best of our knowledge, all existing results on spatial alarms are based on the Euclidean space. This chapter is the first one that develops efficient algorithms and optimizations for scaling road network-aware spatial alarm processing.

5.7 Conclusion

We have presented RoadAlarm – an efficient and scalable approach to processing road network-aware spatial alarms. By utilizing spatial constraints on road networks and mobility patterns of mobile objects, the RoadAlarm approach can provide longer hibernation time of mobile clients while ensuring high success rate. Concretely, we introduce the concept of road network-aware spatial alarms as star-shaped subgraphs and we use the border points to represent the boundary of road network-aware spatial alarms. We design the RoadAlarm baseline algorithm that combines subscription filter with Euclidean Lower Bound (ELB) filter to reduce the search space and speed up the shortest path computation. By further exploring the steady motion-based mobility patterns of mobile objects traveling on a road network, we develop five motion-aware filters: current direction-based filter, destination-based filter, caching-based filter, shortest path-based filter, and selective expansion-based filter. Each improves the previous one by introducing further reduction of border points examined by the RoadAlarm baseline algorithm. Our experiments show that the shortest path-based motion-aware filter can provide three times longer hibernation time than the Euclidean space-based approach and 40% longer hibernation time than the RoadAlarm baseline approach while ensuring high success rate.

CHAPTER VI

WHEN TWITTER MEETS FOURSQUARE: TWEET LOCATION PREDICTION USING FOURSQUARE

The continued explosion of Twitter data has opened doors for many applications, such as location-based advertisement and entertainment using smartphones. Unfortunately, only about 0.58 percent of tweets are geo-tagged to date. To tackle the location sparseness problem, this chapter presents a methodical approach to increasing the number of geo-tagged tweets by predicting the fine-grained location of those tweets in which their location can be inferred with high confidence. In order to predict the fine-grained location of tweets, we first build probabilistic models for locations using unstructured short messages tightly coupled with semantic locations. Based on the probabilistic models, we propose a 3-step technique (Filtering-Ranking-Validating) for tweet location prediction. In the filtering step, we introduce text analysis techniques to filter out those location-neutral tweets, which may not be related to any location at all. In the ranking step, we utilize ranking techniques to select the best candidate location for a tweet. Finally, in the validating step, we develop a classification-based prediction validation method to verify the location of where the tweet was actually written. We conduct extensive experiments using tweets covering three months and the results show that our approach can increase the number of geo-tagged tweets 4.8 times compared to the original Twitter data and place 34% of predicted tweets within 250m from their actual location.

6.1 Introduction

With the continued advances of social network services, such as Twitter, Facebook and Foursquare, a tremendous amount of unstructured textual data has been generated. One of the most popular forms of such unstructured texts is a short text message, called *tweet*, from Twitter and each tweet has up to 140 characters. Twitter users are posting tweets about almost everything from daily routine, breaking news, score updates of various sport

events to political opinions and flashmobs [64, 109]. Over hundreds of millions of such tweets are generated daily. Furthermore, more and more business organizations recognize the importance of Twitter and provide their customer services through Twitter, such as receiving feedback about products and responding to customers' questions using tweets [20].

Tweets can be much more valuable when tagged with their location information because such geo-tagged tweets can open new opportunities for many applications. For example, if a user posts a tweet tagged with her current location, nearby local stores can immediately send her customized coupons based on the context of the tweet or her profile assuming that she is a subscriber of such location-based advertisement services. Similarly, local news and places of interest can be recommended based on the location, the context of the tweet, and the past experiences of her friends in a social network. Geo-tagged tweets can also be used to report or detect unexpected events, such as earthquakes[99], robbery or gun shots, and notify the event to the right people instantly, including those who are close to the location of the event.

On one hand, like most social network services, Twitter recognizes the value of tagging tweets with location information and provides the geo-tagging feature to all its users. On the other hand, such opt-in geo-tagging feature is confronted with several challenges. First, Twitter users have been lukewarm in terms of adopting the geo-tagging feature. According to our recent statistical analysis over 1 billion tweets spanning three months, only 0.58% tweets have their fine-grained location. With such a tiny amount of geo-tagged tweets, it would be very hard to realize the many social and business opportunities such as those mentioned above. Second, even for the limited tweets tagged with geometric coordinates, a fair amount of them cannot be used effectively because their geometric coordinates cannot be served as quality indicators of useful semantic locations, such as points of interest and places where events of interest may happen or have happened. This location sparseness problem makes it very challenging for identifying the types of tweets in which we can infer their location information, i.e., the location where a tweet was written. We argue that in order to derive new values and insights from the huge amount of tweets generated daily by Twitter users and to better serve them with many location-based services, it is important

to have more geo-tagged tweets with semantically meaningful locations.

In this chapter we present a methodical approach to increasing the number of geo-tagged tweets by predicting the fine-grained location of *each tweet* using a multi-source and multi-model based inference framework. Our focus is to predict the location of carefully selected tweets in which their location can be inferred with high confidence based only on their textual data, instead of trying to predict the location of all (or most) tweets. First of all, we address the location sparseness problem of Twitter by building the probabilistic models for locations using unstructured short messages that are tightly coupled with their semantic locations. In order to achieve the tight coupling between text and location, we propose to use Foursquare - a popular location-centric social network, as a source for building these probabilistic models. Based on the probabilistic models, we propose a 3-step technique (Filtering-Ranking-Validating) for predicting the fine-grained location of tweets. In the *filtering* step, we develop a set of filters that can remove those location-neutral tweets, which may not be related to any location at all, prior to entering the location prediction (ranking) phase. This effort enables us to filter out as many location-neutral tweets as possible to minimize the noise level and improve the accuracy of our location prediction model. In the *ranking* step, candidate locations for each tweet are determined using one of the three ranking techniques: standard machine learning approaches, naive Bayes model, and $tfidf$ value. Once the top ranked location is assigned to the tweet, in the *validating* step, we utilize a classification-based prediction validation method to accurately predict the location where the tweet was actually written. We report our experimental evaluation conducted using a set of tweets, collected over a three-month period in New York City. The results show that our approach can increase the number of geo-tagged tweets 4.8 times compared to the original Twitter data and place 34% of predicted tweets within 250m from their actual location.

6.2 Related Work

We categorize the related work into four categories: 1) location prediction in Twitter-like social networks, 2) topic and user group prediction in Twitter-like social networks, 3)

analysis of Foursquare check-ins, and 4) location prediction using other online contents.

Location prediction in social networks. Existing work can be divided into the problem of predicting the location of *each Twitter user* [36, 51, 79] or predicting the location of *each tweet* [57, 73]. Concretely, [36] proposes a technique to predict the *city-level* location of each Twitter user. It builds a probability model for each city using tweets of those users located in the city. Then it estimates the probability of a new user being located in a city using the city's probability model and assigns the city with the highest probability as the city of this new user. To increase the accuracy of the location prediction, it utilizes local words and applies some smoothing techniques. [51] uses a Multinomial Naive Bayes model to predict the *country* and *state* of each Twitter user. It also utilizes selected region-specific terms to increase the prediction accuracy. [79] presents an algorithm for predicting the home location of Twitter users. It builds a set of different classifiers, such as statistical classifiers using words, hashtags or place names of tweets and heuristics classifiers using the frequency of place names or Foursquare check-ins, and then creates an ensemble of the classifiers to improve the prediction accuracy. These coarse-grained location prediction methods rely heavily on the availability of a large training set. For example, the number of tweets from the users in the same city can be quite large and comprehensive. In contrast, the goal of our work is to predict the fine-grained location of each tweet if the tweet can be inferred with high confidence.

[57] and [73] are the most relevant existing work as they centered on predicting the location of each tweet. [73] builds a POI (Place of Interest) model, assuming that a set of POIs are given, using a set of tweets and web pages returned by a search engine. For a query tweet, it generates a language model of the tweet and then compares it with the model of each POI using the KL divergence to rank POIs. Since it uses only 10 POIs and a small test set for its evaluation, it is unclear how effective the approach is in a real-world environment in which there are many POIs and a huge number of tweets and furthermore many tweets contain noisy text, irrelevant to any POI. [57] extracts a set of keywords for each location using tweets from location-sharing services, such as Foursquare check-in tweets, and other general expression tweets posted during a similar time frame. To predict the location of

a new tweet, it generates a keyword list of the tweet and compares it with the extracted keywords of locations using cosine similarity. An obvious problem with this work is that it treats all tweets equally in the context of location prediction. Thus, it suffers from high error rate in the prediction results, especially for those location-neutral tweets.

Topic and user group prediction in social networks. In addition to location prediction of Twitter data, other research efforts have been engaged in inferring other types of information from Twitter data. [75] proposes a framework to predict topics of each tweet. It builds a language model for each topic using hashtags of tweets and evaluates various smoothing techniques. [92] proposes a social network user classification approach, which consists of a machine learning algorithm and a graph-based label updating function. [29] proposes an approach to predict sentiments of tweets and [32] presents a technique to classify Twitter users as either spammers or nonspammers. Most of the work in this category build their language-based classification model using supervised learning and utilize some external knowledge to initialize the classification rules, such as spam or non-spam. In contrast to this line of work, we focus on location detection of tweets rather than Twitter user classification.

Analysis of Foursquare check-ins. [37, 87] analyze Foursquare check-in history in various aspects. [37] shows spatial and temporal (daily and weekly) distribution of Foursquare check-ins. It also analyzes the spatial coverage of each user and its relationship with city population, average household income, etc. [87] also shows spatio-temporal patterns of Foursquare check-ins and calculates the transition probabilities among location categories.

Location prediction using other online contents. Many studies have been conducted to infer the geographical origin of online contents such as photos [105], webpages [24] and web search query logs [59]. [105] builds a language model for each location (a grid cell) using the terms people use to describe images. [24] identifies geographical terms in webpages using a gazetteer to infer a geographical focus for the entire page. [59] utilizes a geo-parsing software that returns a list of locations for web search query logs to infer the location of users (at zip code level).

6.3 Overview

In this section we first describe the reference data models for Twitter and Foursquare data. Then we describe how we build the language models for locations of tweets, using short text messages of Foursquare. Finally we outline the design principles and the system architecture of our location prediction framework.

6.3.1 Twitter Reference Model

Twitter is the most representative microblogging service being used widely, from breaking news, live sports score updates, chats with friends (called *followers*) to advertising and customer service by many companies. Twitter data consists of tweets. Formally, a tweet is defined by a user ID, a timestamp when the tweet was posted, and a short text message up to 140 characters. To enrich its data with location information, Twitter provides not only a location field for each user but also a feature for geo-tagging *each tweet* [8]. Therefore each tweet can be tagged with a fine-grained location, such as a geometric coordinate defined by a latitude and longitude, though the number of tweets with the geo-tag is very small. Our prediction framework performs the location prediction solely based on the short unstructured text messages without requiring user ID and timestamp of tweets. In order to perform text analysis over all tweets, we formally model each tweet as a vector of words in our word vocabulary of n words, denoted by $< w_1, w_2, \ldots, w_n >$. For each tweet tx, if w_1 appears 2 times in tx, we have a value 2 in the position of w_1. Thus, a tweet vector is a vector of n elements of integer type with each element tx_i $(1 \leq i \leq n)$ denoting the number of occurrences of the word w_i in tx. To get a list of words from tweets, we process each tweet by breaking the tweet into tokens, stemming the tokens, and removing stop words from them.

6.3.2 Foursquare Reference Model

Foursquare is a social network service, which is specialized in location-sharing through check-ins. As of May 2014 [2], there are over 50 million users and over 6 billion check-ins, with millions more every day. Users can check into a place by selecting one of the nearby places

from their current location (usually using their smartphones with GPS), and leave tips for a specific place. Each tip has up to 200 characters and is explicitly associated with one place. Foursquare provides the basic information of places, such as name, address, website URL, latitude and longitude, and category. A fair number of Foursquare users are linking their Foursquare account with their Twitter account such that their check-ins are automatically posted to their Twitter account. We argue that building probabilistic language models for locations using Foursquare tips will be the first step towards developing a methodical approach to high quality location prediction for each tweet. Concretely, in order to integrate Foursquare as an external location-specific data source for predicting the location of each tweet, we formally model each tip in Foursquare based on our Twitter vocabulary of n words. Thus, a tip *tip* is also represented as a vector of n elements of integer type, with each element tip_i denoting the number of occurrences of the word w_i in *tip*. Each tip is also associated with a location l. Similar to tweet tokenization process, we get a list of words from tips by breaking each Foursquare tip into tokens, stemming the tokens, and removing stop words from them.

6.3.3 Location Modeling

In contrast to many existing approaches [36, 51, 79, 73, 57], which mainly use geo-tagged tweets to build a probabilistic model for each location, we argue that a high quality location model for tweets should identify those geometric coordinates that are actually associated with some semantically meaningful place(s) of interest (PoI) and build the location models only for those semantic locations, instead of building a location model for every geometric coordinate captured by some tweets. For example, there are many tweets that are not related to any location at all since people can tweet anything regardless of their location. We refer to those tweets that do not relate to any semantic location at all as *location-neutral* tweets. Clearly, if too many such location-neutral tweets are involved in location modeling, the language models we build for locations can be both noisy and misleading. Alternatively, if we counter the sparseness problem of geo-tagged tweets by dividing the geographical region of interest into multiple partitions (such as grids) and then building a

language model using tweets generated in each partition, it will also be misleading since each partition may include tweets from multiple locations and it is hard to differentiate tweets written in one location from those written in another location because each geo-tagged tweet has only latitude and longitude. This problem can be aggravated by the sampling errors existing in most of the localization techniques.

Foursquare, as a location-sharing social network service, has a collection of PoIs (places of interest), and each tip is associated with a short text message and a PoI. This makes Foursquare a valuable resource for building good probabilistic language models for locations, because Foursquare data includes one of the best human-encoded mappings of geometric locations to semantic locations (PoIs) as well as a set of short messages (tips) for them. This motivates us to use Foursquare tips instead of noisy tweets to build more accurate and dependable probabilistic models for locations. In the situation where multiple locations have the same latitude and longitude (such as multistory buildings), we can build a separate language model for each location based on the corresponding PoIs and the set of tips associated with the PoIs.

Let the set of locations (PoIs) in Foursquare be $l_1, l_2, \ldots, l_m$. To predict the location of tweets using the probabilistic models of locations, we first build a language model (LM) for each Foursquare location using a set of tips associated to that location. The language model has a probability for each word (unigram model) or each sequence of n words (n-gram model). Let $tf(w,t)$ denote the number of occurrence of word w in the tip t, $c(w,l)$ denote the number of occurrences of word w in all tips associated to location l and n be the number of all words in our word vocabulary. We calculate the probability of a word w in a location l using the frequency-based maximum likelihood estimation as follows:

$$p(w,l) = \frac{c(w,l)}{\sum_{i=1}^{n} c(w_i,l)}, \quad c(w,l) = \sum_{tip \in tips(l)} tf(w,tip)$$

where $tips(l)$ is the set of tips associated to location l. Given that there are some Foursquare locations with a very small number of associated tips, in order to generate dependable LMs using a sufficient number of tips, we build LMs only for locations with more than a minimum number of tips, defined by a system-supplied parameter θ_{tip} and also consider

only commonly used words in modeling each location.

Bigram Language Model. Instead of the unigram models, where the language model has a probability for each word, we can define a probability for each sequence of n words (n-gram model). For presentation brevity, we below present a bigram model, which can be easily extended to n-gram models. Let $p(w_{i-1}w_i, l)$ be the probability of a bigram $w_{i-1}w_i$ in the tips of location l. The probability of a location l for a tweet T using the bigram LMs is computed as follows:

$$p(l \mid T) = \prod_{w_{i-1}w_i \in T} p(w_{i-1}w_i, l).$$

To estimate the probability of bigrams by handling unobserved bigrams in the tips, in this chapter, we explore three different smoothing techniques: Laplace smoothing, Absolute discounting, and Jelinek-Mercer smoothing [35]. The three smoothing techniques are defined as follows:

Laplace smoothing, which adds 1 to the frequency count of each bigram. This is defined as follows, where $c(w_{i-1}w_i, l)$ is the frequency count of a bigram $w_{i-1}w_i$ included in the tips of location l:

$$p(w_{i-1}w_i, l) = \frac{1 + c(w_{i-1}w_i, l)}{\sum_{w_i}(1 + c(w_{i-1}w_i, l))}$$

Absolute Discounting, which includes interpolation of bigram and unigram LMs by subtracting a fixed discount D from each observed bigram. This is defined as follows, where $N_{w_{i-1}}$ is the number of observed bigrams that start with w_{i-1} such that $\mid \{w_i : c(w_{i-1}w_i, l) > 0\} \mid$:

$$p(w_{i-1}w_i, l) = \frac{max\{c(w_{i-1}w_i, l) - D, 0\}}{\sum_{w_i} c(w_{i-1}w_i, l)} + \frac{D \cdot N_{w_{i-1}}}{\sum_{w_i} c(w_{i-1}w_i, l)} \cdot \frac{c(w_i, l)}{\sum_{w_i} c(w_i, l)}$$

Jelinek-Mercer smoothing, which linearly interpolates between bigram and unigram LMs using parameter λ:

$$p(w_{i-1}w_i, l) = \lambda \cdot \frac{c(w_{i-1}w_i, l)}{\sum_{w_i} c(w_{i-1}w_i, l)} + (1 - \lambda)\frac{c(w_i, l)}{\sum_{w_i} c(w_i, l)}$$

Intuitively, the unigram LMs might be sufficient for short text messages like tweets. However, we will conduct experiments to compare the unigram models with the bigram models in terms of the prediction precision and errors.

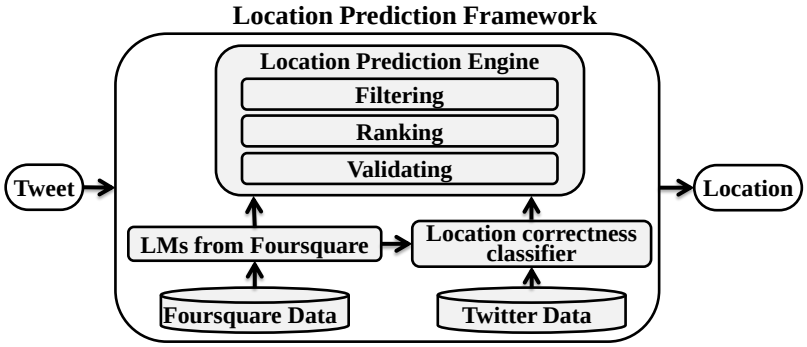


Figure 43: Framework Architecture

6.3.4 System Architecture

Even though we build dependable language models for locations using Foursquare tips, there are still several unique challenges for prediction of the fine-grained location of each tweet. The first challenge is that there are lots of tweets that may not be related to any location at all. Thus, it is important to distinguish those location-neutral tweets, which are completely irrelevant to any location, from those tweets whose locations can be learned and predicted. For example, some daily mundane tweets, such as "`Have a good day!`", rarely have any hint that can be used to predict their location. To address this we need to develop effective techniques to filter out as many location-neutral tweets as possible to minimize the noise level and improve the accuracy of our location prediction model. The second challenge is that a tweet can refer to another location, which is not related to the current location where the tweet was written. For example, it is not unusual that Twitter users post tweets about sports games of their favorite teams even though their current location is not at all related to the locations where the games are being played. Therefore, we also need to develop an approach to detect whether the *referred* location of a tweet, predicted by the location prediction model, is the same as its current location. The referred location of a tweet means the location, which is explicitly mentioned or implicitly hinted in the tweet. Finally, to respect the privacy of users, the location prediction model should not depend on user ID and timestamp of the tweets. To address these challenges, we develop a multi-phase location prediction framework that utilizes the probabilistic models of locations built using Foursquare tips.

Fig. 43 provides a sketch of our system architecture for predicting the fine-grained location of a tweet. Our location prediction engine consists of three steps: (i) **Filtering**: Identification of "I don't know" tweets, which are also referred to as *location-neutral* tweets, (ii) **Ranking**: Ranking and predicting the referred location of a tweet, which is implied explicitly or implicitly by the text message of the tweet, and (iii) **Validating**: Using the classification model to determine whether there is a match between the *referred location* and the *actual physical location* of that tweet. The filtering step is to identify if a tweet has any location-specific information. Our solution approach uses simple and yet effective pruning techniques to differentiate tweets with location-specific information from tweets having no location-specific hint at all, by utilizing the probabilistic language models for locations built using Foursquare tips (Recall the previous section). This allows us to filter out noisy tweets at early phase of the location prediction process. For those tweets that have passed the filtering step, the ranking step is to select the best matched location among the set of possible locations for each tweet using ranking techniques. Finally, the validating step is to validate whether the predicted location of a tweet is indeed the correct location with respect to the actual location where the tweet was written. We will explain each step in detail in the next section.

6.4 Location Prediction

In this section, we describe the key steps we take to predict the fine-grained location of each tweet and how we utilize the probabilistic language models built based on Foursquare tips and the geo-tagged tweets from Twitter in our location prediction framework. We first discuss how to identify and prune the "I don't know" tweets in the filtering step, and then we describe how we design the ranking algorithms to select the best location candidate among a set of possibilities for a tweet in the ranking step. Finally, we discuss how to utilize SVM classifier and the geo-tagged tweets as the training data to develop classification models that validate the correctness of the predicted location of a tweet with respect to the actual physical location from where the tweet was generated, in the validating step.

6.4.1 Filtering Step

We first define "I don't know" tweets as those that have little information about their location or are talking about past or future event. Given a tweet, if there is not any hint about its location, we filter the tweet out because we have no chance of predicting its location using only textual information of the tweet. Also, if a tweet is talking about past or future activities or events, we exclude the tweet because we cannot predict its current actual location even though we may infer the past or future location referred in the tweet. In this chapter, the current location of a tweet refers to a location where the tweet was written. To find such "I don't know" tweets, we utilize local keywords and PoS (Part of Speech) tags.

Utilizing local keywords. Even though each Foursquare tip is explicitly coupled with a location, it also includes some words that are too general to represent the location (e.g. "awesome", "menu", "special"). If a tweet consists of only such general words, it would be impossible to predict the tweet's location because many locations have such words and it is hard to differentiate (rank) among the locations. For example, a tweet "`This sun is BLAZING and there's no shade`" has no hint about its fine-grained location because all words in the tweet are too general to represent any location. To extract any hint about fine-grained locations from tweets, we define *local keywords* as a set of words that are representative of a location. To find the local keywords, we calculate the $tfidf$ (Term Frequency, Inverse Document Frequency) [81] score for each word and each location. Let L be the total number of locations and df_w be the number of locations having w in their tips. Our $tfidf$ calculation for a word w and a location l is formally defined as follows:

$$tfidf_{w,l} = p(w,l) \times \log_{10} \frac{L}{df_w}.$$

For a word w, if there is any location l in which its score $tfidf_{w,l}$ is larger than a threshold, denoted by θ_{tfidf}, we treat the word w as a local keyword with respect to the location l. If a tweet has no local keyword at all, then we classify the tweet as a "I don't know" tweet. The threshold θ_{tfidf} for choosing local keywords is a tuning parameter in our framework. If we increase the threshold value, a smaller number of local keywords will be selected, and

then more tweets could be filtered out as "I don't know" tweets.

Utilizing PoS tags. Even though a tweet has a sufficient number of local keywords, we may not guarantee that the predicted location based on the language models will match the current location with high confidence when the tweet is talking about the future or past event. For example, a tweet "`I'm going to MoMA`" has a local keyword "MoMA" (abbreviation for the Museum of Modern Art in New York City), but is talking about the future location. Therefore, even though we can predict the referred location in the tweet based on the local keywords such as "MoMA" in this example, the predicted location is related to the location where the author of the tweet will be, rather than the current location where this tweet is written. To detect those tweets talking about the past or future location, we utilize PoS (Part-of-Speech) tags generated by a PoS tagger. Given a tweet, if the generated PoS tags of the tweet include any tag about the past tense form, we treat the tweet as a "I don't know" tweet. Since there is no tag about the future tense in existing PoS taggers, we utilize some words related to future or with future sense, such as "will", "going to" and "tomorrow", and remove those tweets that contain such words.

6.4.2 Ranking Step

After filtering out those location-neutral tweets, we explore three different techniques to rank locations for each of the tweets survived from the filtering step. Given a query tweet, there is a set of candidate locations that are associated to the tweet based on the language models for locations. To predict the location of the tweet, we need to rank all locations and select the location having the highest rank (or top k locations) as the predicted location of the tweet.

Standard Machine Learning Approaches. A most intuitive baseline approach is to build classification models using standard machine learning techniques such as SVM and decision tree. To choose a training set for learning the models, we sample some tips for each location. In our training set, each instance and each feature represent a Foursquare tip and a word respectively. The number of classes in the training set is equal to the number of all locations. Thus, given a tweet, we use the predicted class by the classification models as

the predicted location of the tweet.

Naive Bayes Model. Alternatively, given a set of candidate locations for a tweet, we use the simple naive Bayes probabilistic model to rank locations based on the conditional independence assumption among words. Concretely, given a tweet T and the set of possible locations, we calculate the naive Bayes probability for each location l as follows:

$$p(l \mid T) = \frac{p(l) \prod_{w \in T} p(w, l)}{\sum_i p(l_i) \prod_{w \in T} p(w, l_i)}$$

where $p(l)$ is $\frac{1}{L}$ for all locations since in our current implementation we assume the uniform distribution for locations. We predict the location having the highest probability as the tweet's location. To remove any zero probability, we apply Laplace smoothing.

***tfidf* Value.** The naive Bayes model uses the probability of a word in each location when calculating the ranking probability of locations. If we want to reflect how important a word is in *all* locations, we can incorporate such global word weights by using the $tfidf$ values to rank the locations for a given tweet. Concretely, for a given tweet T, let L_T denote the set of candidate locations of T. We calculate the $tfidf$ value for each location l in L_T as follows:

$$tfidf_{T,l} = \frac{\sum_{w \in T} tfidf_{w,l}}{\sum_{l \in L_T} \sum_{w \in T} tfidf_{w,l}}.$$

We use the location having the largest normalized $tfidf$ ranking score as the predicted location of tweet T.

6.4.3 Validating Step

Even though we can filter out some "I don't know" tweets using the local keyword filter and the PoS tag filter, the predicted location for a tweet may not be the actual location where the tweet was written. This is especially true for those tweets whose actual locations where the tweets were written are quite different from the referred location produced by our ranking algorithms. For example, we may think that the referred location in a real tweet "`Let's Go Yankees!!!`" is "Yankees Stadium" and some of our ranking techniques also find "Yankees Stadium" as the predicted location of the tweet. However, it is not unusual that many

New York Yankees fans in the world post such tweets anywhere during the game or before the game. Another interesting real tweet is "`I hope you all have a GREAT weekend but also take time to remember those we've lost; those who are still fighting for our freedom!!`". Under an assumption that we know this tweet is from New York City, some of our ranking techniques find "World Trade Center" as the predicted location of the tweet. We can easily see that the tweet is closely related to "World Trade Center" semantically, however such tweets can be posted from anywhere. The main challenge for predicting the location for this type of tweets is to provide the prediction validation capability for the system to determine if the referred location $l_{ref}(T)$ for a tweet T, obtained using the probabilistic language models and one of the three ranking algorithms, will match the actual location $l_{cur}(T)$ where the tweet T was written. If we detect that $l_{ref}(T)$ does not match $l_{cur}(T)$, then we classify the tweet as an "unpredictable" tweet and exclude the tweet from our location prediction.

Our approach to finding such "unpredictable" tweets is to build a classification model using standard machine learning techniques. To learn the classification model, we need to prepare a training set carefully. One approach to preparing the training set is to use those tweets having a geo-tag (i.e., latitude and longitude), because such tweets already have their explicit current location, thus we can use the language models and one of the ranking algorithms to extract their referred location to build the training set. Given a tweet T having its geo-tag, after choosing the location (denoted as $l_{top}(T)$) having the highest probability based on the naive Bayes probability, we additionally compare the probability of $l_{top}(T)$ with that of the other locations using a probability ratio test. We use this test to build a good training set consisting of only tweets in which there is high confidence in their referred location. We choose only those tweets that pass the probability ratio test, formally defined as follows:

$$\frac{p(l_{ref}(T) \mid T)}{1 - p(l_{ref}(T) \mid T)} > \delta$$

where δ is the criterion of our test. If we increase δ, a smaller number of tweets will be selected for the training set.

Based on the generated training set, we learn classification models by running the decision tree classifier and SVM (Support Vector Machine) with the polynomial kernel functions and Gaussian radial basis functions using 10-fold cross-validation. Then we choose a classification model having the highest cross-validation precision for the training set and use this classification model for detecting the "unpredictable" tweets. To find parameters having the highest cross-validation precision, we use the grid search. We introduce some notable results returned by our classification model. For a real tweet "`The line at this Chipotle in Brooklyn Heights is really long`", our model detects that its referred location, produced by the language models and the ranking algorithm, indeed matches the actual location where this tweet was written, as indicated by the geo-tag of the tweet. Therefore, our model correctly classifies this tweet and thus validates the correctness of our predicted location of the tweet. Note that the accuracy of the prediction depends on our language models whereas the accuracy of the prediction validation depends on the training set.

6.5 Experiments

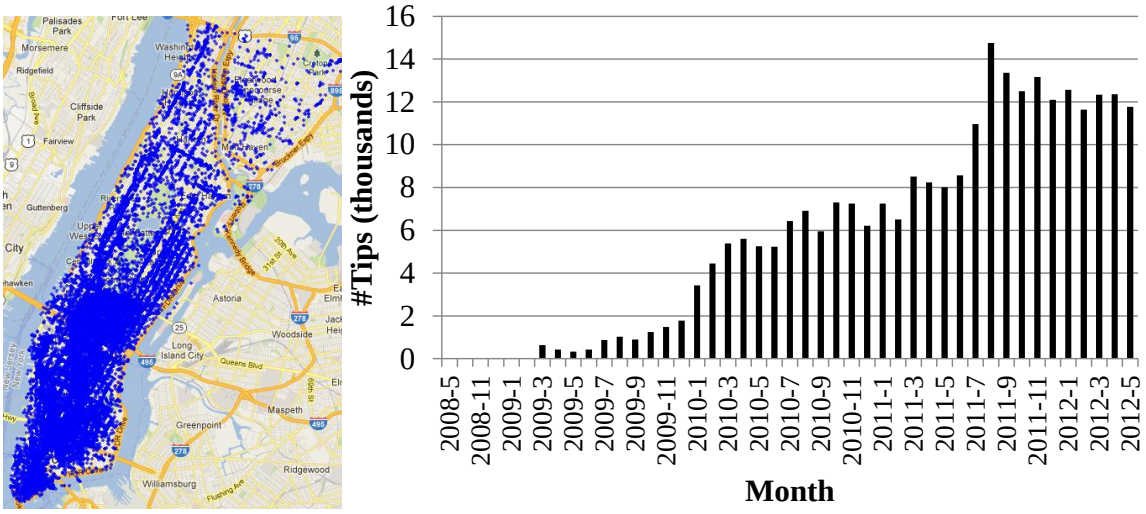
In this section, we evaluate the proposed location prediction framework for tweets through an extensive set of experiments conducted using tweets collected over a three-month period. We report the experimental results on how we build the language models using the datasets, how we implement the prediction validation classifier to distinguish the predictable tweets from those non-predictable ones, and the effectiveness of the two filters to find "I don't know" tweets. In addition, we evaluate the effectiveness of our location prediction approach by studying the effects of different parameters on the precision of location prediction, such as the effects of different ranking methods, the effects of unigram v.s. bigram language models, the effects of different δ values for building prediction validation classifier, and the effects of different $tfidf$ threshold values.

6.5.1 Datasets

We gathered a set of tweets spanning from April 2012 to June 2012 using Twitter Decahose [21], which is a feed of 10% of all tweets. Each day (24 hours) has about 37 million

tweets and only 0.58% tweets are geo-tagged (i.e., include fine-grained location information). To focus on predicting the fine-grained location, we assume that we know the city-level (or similar) location of tweets because previous work [36, 79] has addressed this. Since some tweets explicitly include their city-level location even though they don't have their geo-tag, we can also utilize such information. In this chapter, we select tweets from Manhattan, New York, USA because Manhattan (officially a borough of New York City), which covers 59 square kilometers (23 square miles), is one of the biggest and most densely populated cities in the world. Based on their geo-tag (latitude and longitude), 127,057 tweets (spanning three months) from Manhattan are selected. Among them, we exclude 39,157 tweets from Foursquare and 15,299 tweets from Instagram to remove any possible bias from them because they already include the location name in their textual data and so it would be straightforward to predict their location. Therefore, we use 72,601 tweets to evaluate our prediction framework.

We extracted Foursquare locations, called venues, and their tips using Foursquare API. First, to gather a set of Foursquare locations, we called the Foursquare venues API for each cell after splitting the area of Manhattan into very small cells (each covers 50 m $\times$ 50 m). Unfortunately, there were some missing locations using only this grid search. Therefore, to find additional locations, we analyzed the URLs included in check-in tweets from Foursquare and then extracted location IDs from them. Each Foursquare location has basic information such as name, address, latitude, longitude, city, country and category. Finally, for each gathered location, we extracted all its tips using Foursquare API. Using this approach, we gathered 25,171 venues in Manhattan and their 268,470 tips, which span from May 2008 to June 2012. Also, there are some locations in which their area is too wide to represent their location using only one point, such as Central Park, Times Square and Yankee Stadium. Since Foursquare does not provide boundary information of its locations, we extracted boundary information of 22 wide locations in Manhattan using Google Maps. Fig. 44(a) shows the geographical distribution of Foursquare locations in Manhattan and Fig. 44(b) shows the distribution of total tips over the past 4 years, which shows a tremendous increase in the number of Foursquare tips in the last year.



(a) Locations in Manhattan

(b) #Tips in Manhattan by month

Figure 44: Foursquare Locations and Tips

6.5.2 Building Language Models

To build our language models for the extracted locations, we first choose locations that have more than 50 tips and so 1,066 locations are selected. We also experimented using language models of locations having more than 30 tips and 100 tips. However, the location prediction accuracy using them was not better than using locations having more than 50 tips. We believe that 30 or 40 tips are not enough to build a distinct language model for each location. On the other hand, for locations having more than 100 tips (e.g., 500 tips), we believe that the prediction accuracy will improve with more tips. However, there are only about 300 Foursquare locations in Manhattan having more than 100 tips and we think this number is too small to cover the area of Manhattan. Therefore, in this chapter, we report results using language models of locations having more than 50 tips. For each location, to get a list of words from its tips, we first break each tip into tokens. Then we stem the tokens using Snowball stemmer [18] and remove any stop words in the tokens using stop words of Rainbow [82]. In addition to removing stop words, to consider only commonly used words for the location, we exclude words that appear in less than 5% tips among all tips of the location. Through this filtering, we can remove those words that are less common or contain typos, thus reduce the size of our word vocabulary (i.e., a set of all words used in

our language models). Finally, 3,073 words are included in our word vocabulary.

6.5.3 Finding "I don't know" Tweets

To find local keywords, we empirically choose three different $tfidf$ threshold values: 0.1, 0.2 and 0.3. For example, let us assume that a word appear in 10% of all locations (i.e. inverse document frequency, $idf = 1$). We can intuitively think that the word is too general to be included in the local keywords. By using 0.1 as the threshold, there should be any location in which the term frequency (tf) of the word is larger than 0.1 to be selected as a local keyword. Since it is rare for a word to occupy 10% of all tips, the word will be filtered out by the threshold. Table 15 shows the number of selected local keywords, among 3,073 words in our word vocabulary, for different $tfidf$ threshold values. To find tweets which are talking about the future or past, we utilize PoS tags generated by GPoSTTL [10].

Table 15: Local Keywords

$tfidf$ **threshold**	**# local keywords**
0.1	1,782
0.2	556
0.3	200

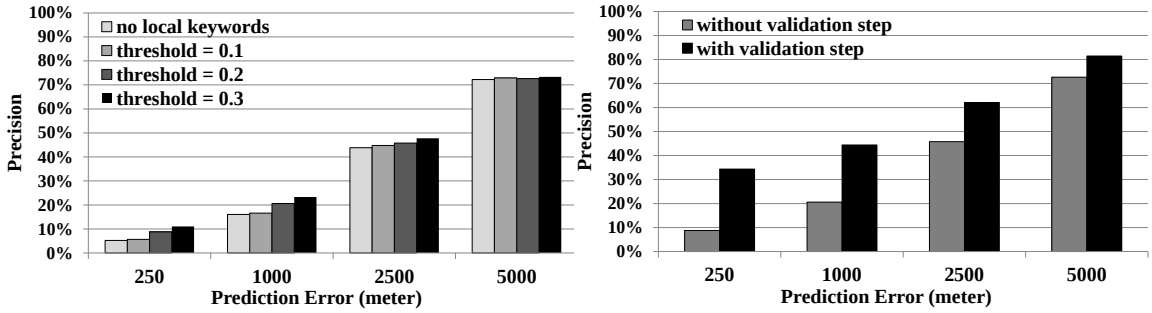
6.5.4 Prediction without the Validating Step

First we evaluate the prediction accuracy of our framework without applying the validating step for the predicted locations. To measure the prediction accuracy, given a tweet, we compare the geo-tag, which was removed during the prediction steps, of the tweet with the latitude and longitude (or boundary) of the predicted location. If the predicted location has its boundary information and the geo-tag of the tweet is within the boundary, the prediction error is 0. Otherwise, we calculate the Euclidean distance between the geo-tag of the tweet and the latitude and longitude of the location and then use the distance as the prediction error. We also note that acceptable prediction errors depend on the application in question. For example, automated geospatial review applications may require the location of the individual to be identified accurately (within 100m). On the other hand, applications such as event localization can tolerate a few hundreds of meters of error.

Table 16 shows that our framework without the validating step can geo-tag a much

Table 16: Geo-tagged Tweets without the Validating Step

tfidf **threshold**	**# geo-tagged tweets**	**percentage**
No local keywords	31,264	43.06%
0.1	28,057	38.65%
0.2	15,096	20.79%
0.3	7,168	9.87%



(a) Without the validating step (b) Effects of the validating step (threshold: 0.2)

Figure 45: Effects of the Validating Step

more number of tweets, compared to 0.58% in the original Twitter data. However, as shown in Fig. 45(a) where we use the naive Bayes model as the ranking technique (we will compare different ranking techniques in the next section), the prediction precision is not satisfactory because only 10% of predicted tweets are located within 250m from their actual location even though we apply very selective local keywords (i.e., threshold = 0.3). Here, the precision means the percentage of predicted tweets whose prediction error is less than a specified distance (250m, 1,000m, 2,500m and 5,000m in Fig. 45(a)). Although this result is meaningful compared to existing coarse-grained prediction frameworks, one of our goals is to improve the accuracy of our predicted locations. The results in subsequent sections show that we can considerably improve the prediction accuracy using our validating step.

6.5.5 Building Models for the Validating Step

To validate the correctness of the predicted locations in terms of their likelihood to match the actual location where the tweets were written, we need to learn our classification models using the training datasets. In this set of experiments, we empirically use three different δ values: 0.5, 1.0 and 2.0 to generate three training sets. In other words, given a tweet, if there is a location whose naive Bayes probability is larger than 33%, 50% and 66%, the

tweet will be included in the training set with the δ value of 0.5, 1.0 and 2.0 respectively. For each tweet, to label whether its referred location is equal to its current location, we compare the latitude and longitude of the referred location, extracted from Foursquare, with the geo-tag (i.e. current location) of the tweet. If the distance between the two locations is less than 100 meters or the geo-tag of the tweet is within the boundary of its referred location, we label that the tweet's two locations are the same. Table 17 shows the number of selected tweets, the number tweets whose two locations are different and the number of tweets whose two locations are the same, for different δ values among 72,601 tweets.

Table 17: Training Sets

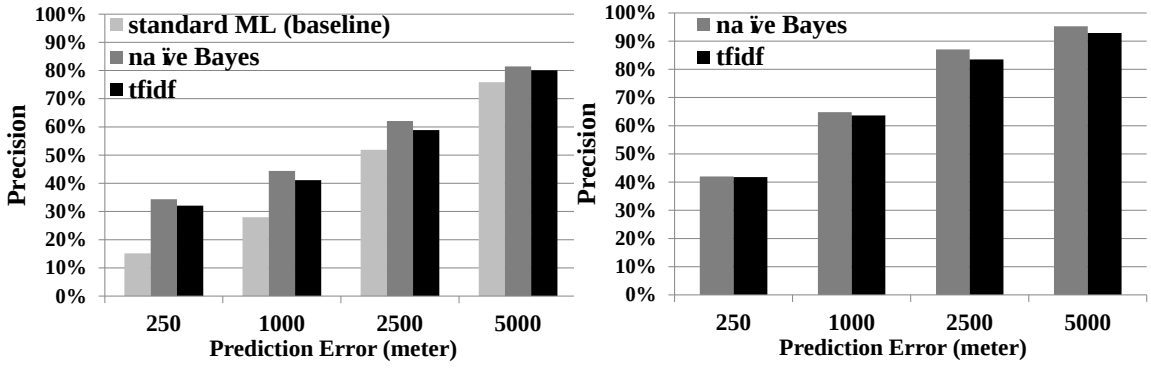
δ **value**	**# tweets**	# $l_{ref} \neq l_{cur}$	# $l_{ref} = l_{cur}$
0.5	2,642	1,936	706
1.0	1,598	1,008	590
2.0	1,028	579	449

6.5.6 Prediction with the Validating Step

In this section, we first show the effectiveness of our classification-based prediction validation step for improving the prediction accuracy. Then we compare the location prediction accuracy by different ranking techniques and different parameter values. In this section, we use the $tfidf$ threshold of 0.2 and the δ value of 0.5, unless specifically noted, because we think this setting strikes a balance between the number of geo-tagged tweets and the prediction accuracy. We will show the effects of different parameter values in this section.

Effects of the validating step. Fig. 45(b) shows that we can significantly improve the prediction precision using our validating step, compared to that without the validating step. Based on the generated classification model, by filtering out those tweets in which their predicted location does not match their actual location, we can locate about 34% of predicted tweets within 250m from their actual location.

Effects of different ranking techniques. Fig. 46 shows the prediction precision of three different ranking techniques on 2003 tweets predicted by our framework. We will show how 2003 tweets are predicted in the next experiment. Fig. 46(a) shows that using the naive Bayes model as the ranking technique has better prediction precision than using standard



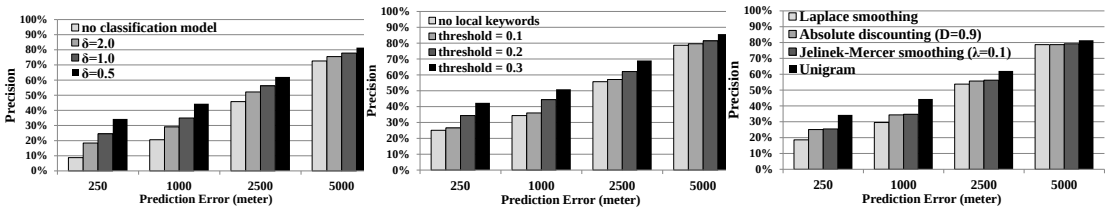
(a) Using only one prediction

(b) Using the best in the Top-5

Figure 46: Effects of Different Ranking Techniques

machine learning techniques (our baseline approach) or $tfidf$ values. Specifically, using the naive Bayes model, about 34.35% and 44.38% of predicted tweets are located within 250m and 1,000m respectively from their location. This result shows that the naive Bayes model is working well in our language models to rank locations for given tweets even though the model does not consider global word weights. We think this is because our language models include only location-specific words (i.e., most of general words are filtered out by our local keywords and stop words). This may also be a reason that incorporating global word weights of such location-specific words, like $tfidf$ ranking, does not help much in terms of improving the prediction precision. In comparison, ranking with the standard machine learning (ML) techniques has relatively worse prediction precision because the prediction model is built using a very limited number of Foursquare tips. Since it is almost infeasible to use all (or most of) tips to run standard ML techniques due to the time complexity and the resource (CPU and memory) constraints, it would be hard to get good prediction results using this technique.

Fig. 46(b) shows the prediction precision using the best prediction (i.e., the closest location from the geo-tag of tweets) in the top-5 predictions. This result represents the capacity of our prediction framework to find a set of good candidate locations even though the first predicted location is mistaken. The result shows that the naive Bayes model also has the best prediction precision by locating 41.99% of predicted tweets within 250m from their location. The prediction model generated using standard ML techniques has no top-5 result



(a) Effects of different δ values (b) Effects of different $tfidf$ threshold values (c) Effects of bigram LMs

Figure 47: Effects of Different Parameter Values

because it returns only one location having the highest confidence. Since the naive Bayes model has the best prediction precision in all other experiments using different parameter values, we report results using only the naive Bayes model in subsequent sections.

Table 18: Effects of Different δ Values

δ **value**	**# geo-tagged tweets**	**percentage**
0.5	2,003	2.76%
1.0	2,764	3.81%
2.0	3,982	5.48%

Effects of different δ values. We compare the number of tweets, among 15,096 tweets (See Table 16), classified as $l_{ref} = l_{cur}$ by different classification models built using different δ values in Table 18. The percentage in the table shows the ratio among 72,601 target tweets. Since the classification model using 0.5 as the δ value is built using the training set which includes more $l_{ref} \neq l_{cur}$ tweets compared to the other training sets as shown in Table 17, it has more capability to find such tweets and so choose fewer predictable tweets. The prediction precision result below shows that the classification model built using the δ value of 0.5 ensures higher precision by effectively filtering out unpredictable tweets. Fig. 47(a) shows the prediction precision of our framework without any classification model and with three different classification models using different δ values. The prediction precision increases as the δ value decreases because, as we mentioned, the capability to filter out $l_{ref} \neq l_{cur}$ tweets increase due to the higher percentage of $l_{ref} \neq l_{cur}$ tweets in the training set. However, there would be a point in which selecting more tweets for learning the classification model by decreasing the δ value does not improve the prediction precision any more (or even worsens the prediction precision). This is because more noisy tweets

that have low confidence in their referred location would be included in the training set by decreasing the δ value.

Effects of different $tfidf$ threshold values. Fig. 47(b) shows the prediction precision of our framework without any local keywords and with three different $tfidf$ threshold values. Since the number of local keywords decreases as we increase the $tfidf$ threshold values as shown in Table 15, more tweets are filtered out as "I don't know" tweets because tweets should have at least one local keyword not to be excluded. Also, the precision continuously increases because selected tweets by high $tfidf$ threshold for the prediction have unique location-specific keywords. However, there is a trade-off between the prediction precision and the percentage of selected tweets. In other words, if we increase the $tfidf$ threshold to improve the prediction precision, a smaller number of tweets are selected for the prediction.

Unigram vs Bigram. In this section we compare unigram and bigram LMs under the same conditions. Fig. 47(c) shows the prediction precision of bigram LMs with three different smoothing techniques and unigram LMs using the naive Bayes model. The effective smoothing parameters are selected from a coarse search of the parameter space. The result shows that unigram LMs are more effective than bigram LMs, which is consistent with the reported results [102]. This is because tweets and Foursquare tips are very short messages and it is rarely possible to include a bigram (or trigram or more), which can be used to effectively differentiate one location from another. Even though the location names include two or more words, the examination of prediction results verifies that unigram LMs are sufficient to detect such names. Furthermore, the effective parameters of absolute discounting and Jelinek-Mercer smoothing show that the smoothed bigram LMs work better when they assign more weights on unigram LMs.

Table 19: Percentage of Geo-tagged Tweets

Approach	**Percentage**
original Twitter data	0.72%
original Twitter data (excluding Foursquare & Instagram)	0.58%
our framework (without validation step)	20.79%
our framework (with validation step)	2.76%

6.5.7 Percentage of Geo-tagged Tweets

Finally we summarize how many tweets are geo-tagged by our prediction framework in Table 19. This result indicates how well our framework tackles the location sparseness problem of Twitter. In the original Twitter data, only 0.72% tweets have their geo-tag. For fair comparison with our framework in which we exclude tweets from Foursquare and Instagram because it is too trivial to predict their location, the percentage of geo-tagged tweets in the original Twitter data goes down to 0.58% if we don't count the tweets from Foursquare and Instagram. We report in this section the results of our framework using the δ and $tfidf$ threshold value of 0.5 and 0.2 respectively and the naive Bayes model as the ranking technique because we think this setting strikes a balance between the number of geo-tagged tweets and the prediction accuracy. Our framework equipped with all proposed techniques including the validating step can geo-tag 2.76% of all tweets, increasing about 4.8 times compared with the percentage of geo-tagged tweets in the original Twitter data, while placing 34% of predicted tweets within 250m from their actual location. If we don't use our classification-based prediction validating method, we can geo-tag 20.79% of all tweets with lower prediction accuracy as shown in Table 16.

6.6 Conclusion

We have addressed the location sparseness problem of Twitter by developing a framework for increasing the number of geo-tagged tweets by predicting the fine-grained location of each tweet using only textual content of the tweet. Our framework is vital for many applications that require more geo-tagged tweets such as location-based advertisements, entertainments, and tourism. Our prediction framework has two unique features. First of all, we build the probabilistic language models for locations using unstructured short messages that are tightly coupled with their locations in Foursquare, instead of using noisy tweets. Second, based on the probabilistic models, we propose a 3-step technique (Filtering-Ranking-Validating) for tweet location prediction. In the filtering step, we develop a set of filters that can remove as many location-neutral tweets as possible to minimize the noise level and improve the accuracy of our location prediction models. In the ranking step, we

utilize ranking techniques to select the best candidate location as the predicted location for a tweet. In the validating step, we develop a classification-based prediction validation method to ensure the correctness of predicted locations. Our experimental results show that our framework can increase the percentage of geo-tagged tweets about 4.8 times compared to the original Twitter data while locating 34% of predicted tweets within 250 meters from their location. To the best of our knowledge, this is the first work, which incorporates external data source such as Foursquare, in addition to Twitter data, for location prediction of each tweet. Furthermore, unlike most existing frameworks that focus on coarse-grained prediction such as 10km and 100km, our framework locates a considerable amount of predicted tweets within one-quarter kilometer from their location.

It should be noted that, for privacy advocates, our results can be interpreted as new threats to location privacy for their short messages such as tweets. In other words, our techniques can be used not only to provide the valuable geo-tag information of tweets for location-based services but also to give warning of potential risks to their location to the privacy advocates. For example, when a Twitter user, who is concerned about his/her privacy, posts a tweet, our framework can detect that the location of the tweet can be predicted with high confidence and give him/her a warning of potential threats to location privacy. Our framework can also provide real-time warnings, while the user is writing a tweet, by checking whether the newly entered word is included in the local keywords.

Even though the focus of this chapter is exploring location-specific information explicitly or implicitly included in the textual content of tweets, our framework can be extended by incorporating more information sources to further increase the number of geo-tagged tweets and improve the location prediction accuracy. One simple extension could be to build time-based models (per day, week, month and year) for each location and then utilize the models with the timestamp of a given tweet to predict its location. For example, if our time-based models for a museum indicate that there is almost no activity after 6pm on weekdays, our prediction framework would give very low ranking to the museum for a tweet that was posted at 9pm on Wednesday. Another possible extension could be to consider a set of tweets, including Foursquare check-in tweets, posted by a single user as time series

data. This information could be used to fine-tune the prediction of our framework. For example, if a user posted a Foursquare check-in tweet, we can reduce the search space for predicting the location of those tweets, posted by the same user and whose timestamp is close to that of the Foursquare tweet. Furthermore, if a user posted two Foursquare check-in tweets at two different locations within a short period of time, we could predict the location of those tweets posted between the two timestamps of the Foursquare tweets by analyzing the possible trajectory paths between the two locations using some interpolation techniques, like the route matching algorithm [112]. Other interesting extensions to our current framework includes inference over future and past activities included in the tweets, utilizing social relationships between Twitter users, spatial and temporal relationship as well as semantic relationship among different tweets.

CHAPTER VII

EFFICIENT SPATIAL QUERY PROCESSING FOR BIG DATA

Spatial queries are widely used in many data mining and analytics applications. However, a huge and growing size of spatial data makes it challenging to process the spatial queries efficiently. In this chapter we present a lightweight and scalable spatial index for big data stored in distributed storage systems. We also extend our spatial index for graph models. Our spatial index has several advantages over existing techniques. First, it can be easily applied to existing storage systems or graph models without modifying their internal implementation. Second, it achieves high pruning power by selecting only relevant spatial objects efficiently based on a simple yet effective filter. For example, even though our index does not construct any complicated data structure, the precision (the ratio of true positives to all evaluated spatial objects) of our index is one order of magnitude higher than that of an R-tree-based index for those range queries having high selectivity. Third, it supports a customizable and intuitive control of index size (i.e., the precision of indexed geometries). Last but not the least, it supports efficient updates of spatial objects because it does not maintain any costly data structure such as trees. Experimental results show the efficiency and effectiveness of our spatial indexing technique for different spatial queries.

7.1 Introduction

Many real-world and online activities are associated with their spatial information. For example, when we make or receive a call, the call information including its cell tower location is stored as a call detail record (CDR). Even a single tweet message of Twitter can be stored with its detailed location (i.e., latitude and longitude) [8]. To extract more valuable and meaningful information from such spatial data, spatial queries are widely used in many data mining and analytics applications. One of the most representative challenges for processing the spatial queries is that the amount of spatial data is increasing at an unprecedented rate, especially thanks to the widespread use of GPS-enabled smart-phones.

Due to this huge size of spatial data, we need new scalable techniques that can process the spatial queries efficiently.

To handle such huge spatial data, it is natural to utilize emerging distributed computing technologies such as Hadoop MapReduce, Hadoop Distributed File System (HDFS) and HBase. Several techniques have been proposed to support spatial queries on Hadoop MapReduce [78, 122, 23, 123] or HDFS [74, 76]. However, most of them require internal modification of underlying systems or frameworks to implement their indexing techniques based on, for example, R-trees. Those approaches not only increase the complexity and overhead of the modified storage systems but also are applicable only to a specific storage system.

If the spatial data is represented as a graph structure, we can execute more complex spatial queries using relationships among spatial objects in the graph. For example, in social networks, instead of simply retrieving all users residing in a certain region, we can extract all pairs of users who not only reside in a certain region but also have at least one common friend and graduated from the same high school, based on the relationships in the graph. A few techniques [93, 69] have been proposed to support spatial queries in graph models. However, since they require internal modification of standard graph models such as Resource Description Framework (RDF), it is hard to integrate them with existing graph management systems.

To tackle the limitations of existing work, in this chapter, we investigate the problem of developing efficient and scalable techniques for processing spatial queries over big spatial data. Specifically, we present a lightweight spatial index based on a hierarchical spatial data structure. Our spatial index has several advantages. First, it can be easily applied to existing storage systems without modifying their internal implementation and thus we can utilize existing systems as they are. Second, it provides simple yet highly efficient filtering, based on prefix matching, for finding only relevant spatial objects. Third, it supports a customizable and easy-to-use control of index size for different applications and thus we can reduce the index size at a cost of pruning power. Last but not the least, it supports efficient updates of spatial objects because it does not maintain any costly data structure such as

trees.

Based on the spatial index, this chapter makes three novel contributions. First, we develop an efficient spatial index for big data stored in a distributed storage system. We demonstrate how we implement the index on top of HBase without modifying its internal implementation. Second, we extend the index for spatial data stored as a graph structure. We present how we implement the index on top of RDF, without modifying the standard model, and a query rewriting technique to support spatial queries using a standard graph query language. Third, we provide experimental results to show the efficiency and effectiveness of our spatial indexing techniques.

The rest of the chapter is organized as follows. We give an overview of spatial queries, hierarchical spatial data structure, distributed storage systems, and graph models and outline the related work in Section 7.2. In Section 7.3, we present our spatial indexing techniques for distributed storage systems and graph models. We evaluate the performance of our spatial index in Section 7.4 and conclude the chapter in Section 7.5.

7.2 Preliminary

In this section, we give an overview of spatial queries, hierarchical spatial data structure, distributed storage systems, and graph models. We also outline the related work.

7.2.1 Spatial Queries

There are many types of spatial queries, such as selection query, join query and k nearest neighbor (kNN) query, for different applications. Even though there are more spatial relations [89], in this chapter, we focus on selected fundamental queries that are basis for many other spatial queries: *containing*, *containedIn*, *intersects*, and *withinDistance*. Those queries are defined for any geometries including points, lines, rectangles, and polygons. A ***containing(search geometry)*** query returns all spatial objects that contain the given search geometry. A ***containedIn(search geometry)*** query returns all spatial objects that are contained by the given search geometry (i.e., the converse of *containing*). An ***intersects(search geometry)*** query returns all spatial objects that intersect with the given search geometry. A ***withinDistance(search geometry, distance)*** query (or range

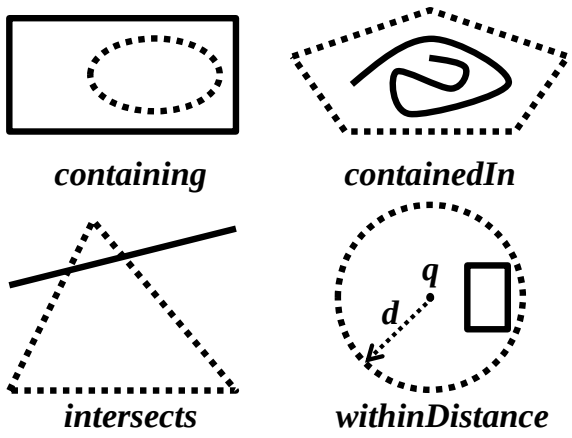


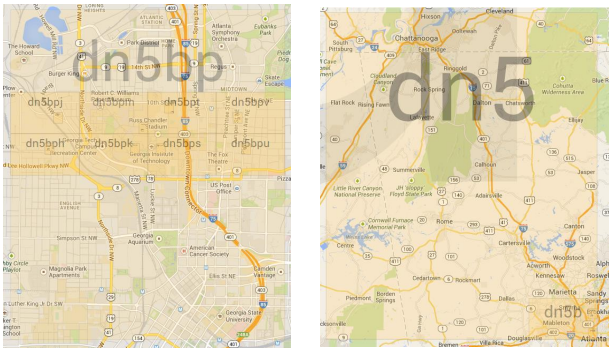
Figure 48: Spatial Queries

query) returns all spatial objects that are within the given *distance* from the the given search geometry. Fig. 48 shows spatial query examples in which, for each query, there is one spatial object satisfying the query condition. Search geometries and spatial objects are represented using dotted lines and solid lines respectively.

7.2.2 Hierarchical Spatial Data Structure

For our spatial indexing, we utilize a hierarchical spatial data structure, called *geohash* [9], which is a geocoding system for latitude and longitude. A geohash code, represented as a string, basically denotes a rectangle (bounding box) on the earth. It provides a spatial hierarchy and it can reduce the precision (i.e., represent a bigger rectangle) by removing characters from the end of the string. In other words, the longer the geohash code is, the smaller the bounding box represented by the code is. Another property of geohash is that two places with a long common geohash prefix are close each other. Similarly, nearby places *usually* share a similar prefix. However, it is not always guaranteed that two close places share a long common prefix.

Definition 21 (Geohash code) Given a geographic location with latitude lat and longitude $long$, the geohash code of the location, denoted by $geohash(lat, long)$, is a sequence of characters $c_1c_2 \ldots c_k$. The geohash code $geohash(lat, long)$ defines a bounding box within which the location lies.



(a) geohash code "dn5bp" (b) geohash code "dn5"

Figure 49: Geohash Examples

Property 1 (Gradual precision) Let $c_1c_2 \dots c_k$ denote a geohash code and $bb(c_1c_2 \dots c_k)$ denote a bounding box represented by $c_1c_2 \dots c_k$. A bounding box represented by any prefix $c_1c_2 \dots c_m$ $(m < k)$ of the geohash code $c_1c_2 \dots c_k$, denoted by $bb(c_1c_2 \dots c_m)$, fully contains $bb(c_1c_2 \dots c_k)$. Conversely, a bounding box represented by a geohash code $c_1c_2 \dots c_k \dots c_n$ $(n > k)$ having $c_1c_2 \dots c_k$ as its prefix, denoted by $bb(c_1c_2 \dots c_k \dots c_n)$, is fully contained in $bb(c_1c_2 \dots c_k)$.

For example, geohash code "dn5bp" covers midtown and downtown of Atlanta, Georgia, USA as shown in Fig. 49(a) [1]. Geohash code "dn5bps" having prefix "dn5bp" represents a smaller region inside midtown Atlanta (i.e., a sub-rectangle of the rectangle represented by "dn5bp"). If we remove two character from the end of the code, geohash code "dn5" represents a much bigger region intersecting three US states (Georgia, Tennessee and Alabama) as shown in Fig. 49(b). The rectangle represented by geohash code "dn5b" is located in the bottom right-hand corner of the rectangle represented by geohash code "dn5".

7.2.3 Distributed Storage Systems

A growing number of non-relational distributed databases (often called NoSQL databases) are proposed and widely used in many big data applications and analytics because they are designed to run on a large cluster of commodity hardware and fault-tolerant through data replication. One representative category of the NoSQL databases is the key-value store,

[1] Generated from http://geohash.gofreerange.com/

in which data is stored in a schema-less way via an unique key that represents each row, such as Apache HBase, Apache Accumulo, Apache Cassandra, Google BigTable, Amazon DynamoDB, just to name a few. In this chapter, our description is based on HBase, an open-source key-value store (or wide column store) originally derived from BigTable, because it is widely used by many big data applications. However, we believe that our spatial index is applicable to other key-value stores similarly because we use only keys for our index without modifying the internal structure of HBase.

A HBase table consists of rows and the rows are stored in sorted order. Each row has a primary key and an arbitrary number of columns. Unlike traditional relational databases, different rows can have different columns. Columns are grouped into column families and the data under the same column family is stored together. HBase usually uses HDFS as its underlying file system and provides random read/write accesses to the data stored in HDFS.

7.2.4 Graph Models

Graph-based data analytics is invaluable because graphs are everywhere from social networks to brain networks and we can extract more meaningful information through structural relationships in the graph. Recently, several single machine-based systems [67, 98] or distributed systems [80, 45] have been proposed to process big graph data. In this chapter, we develop our spatial index on top of the RDF graph model [17], which is a standard model adopted by World Wide Web Consortium (W3C) and widely used in not only research communities but also many governments. An RDF graph consists of RDF triples and each RDF triple has three components: subject, predicate, and object. An RDF triple represents a directed edge, from the subject to the object, having the predicate as its edge label.

SPARQL [19] is a standard query language, adopted by W3C, for RDF graphs. A SPARQL query consists of triple patterns that are similar to RDF triples but its subject, predicate and object can be a variable. Executing a SPARQL query is basically to find a set of subgraphs, satisfying the given graph pattern, where the terms in the subgraphs may be substituted for the variables of the query. For example, the below SPARQL query requests

all users who received their PhD degree at a college called "GT" in 2014.

```
PREFIX gt: <http://cc.gatech.edu/disl/>
SELECT ?user
WHERE {
    ?user gt:phdFrom ?college .
    ?college gt:name "GT" .
    ?user gt:phdYear "2014" . }
```

7.2.5 Related Work

We classify existing spatial query processing techniques using distributed computing frameworks into two categories, based on their query types. The first category handles high selectivity spatial queries, such as selection queries and kNN queries, in which only a small portion of spatial objects are returned as the result of spatial query processing. A few techniques have been proposed to process the high selectivity queries in HDFS [74, 76]. They are utilizing popular spatial indices such as an R-tree and its variants [48, 104, 31]. [74] implements a built-in block-based hierarchical index structure, based on an R-tree, in HDFS to process high selectivity spatial queries. Its R-tree index is stored as a file in HDFS and nearby leaf nodes are stored in the same block to preserve the proximity. [76] combines those small files, which are in adjacent location, into one group in HDFS to reduce the number of HDFS files. Then it builds a hashing-based index for the small files.

The second category handles low selectivity spatial queries that usually require at least one full scan of each dataset. One of the most representative low selectivity spatial queries is k nearest neighbor join (kNN join), which is to find, for each object in a dataset A, its k nearest neighbors in another dataset B. Several techniques have been proposed to process the kNN (or similar) joins using the MapReduce framework [78, 122, 23, 123]. [78] first divides the objects in A into partitions based on a Voronoi diagram with selected pivots. For each partition of A, it finds a subset of B, which includes kNNs of all objects in the partition, using a MapReduce job. [122] basically runs two MapReduce jobs to execute a kNN join, based on the block nested loop methodology. In the first job, it splits each

dataset into n equal-sized blocks in the Map phase and compares every possible pair of blocks (one from A and one from B) to find local kNNs in the Reduce phase. Finding local kNNs in the Reduce phase can be improved by building an R-tree for each local block of B. In the second job, it merges all local kNNs of each object in A and then finds the global kNNs. It also proposes an approximate algorithm, to improve the scalability, which transforms multi-dimensional datasets into one dimension using space-filling curves. [23] first constructs a Voronoi diagram for the given input dataset using a MapReduce job in which it creates partial Voronoi diagrams in the Map phase and combines them into a single Voronoi diagram in the Reduce phase. Based on the Voronoi diagram, it supports three point-based queries: reverse nearest neighbor, maximizing reverse nearest neighbor and kNN. [123] executes the spatial selection query, join query, kNN and all-nearest-neighbors query (ANN) using MapReduce jobs. However, it basically evaluates all objects even for the spatial selection query.

Our work basically belongs to the first category because our focus is to efficiently find a set of spatial objects satisfying the given query. However, as we will explain later, our spatial index can be applied for the second category (i.e., MapReduce jobs) as an efficient and lightweight filtering approach for the input data, instead of reading the whole data regardless of the query conditions.

In terms of graph models, a few techniques [93, 69] have been proposed to support spatial queries for RDF. [93] proposes an extension of SPARQL for complex spatiotemporal queries. It introduces the spatial filter to express spatial constraints such as *inside*, *contains* and *intersect*. To process the spatial queries, it stores RDF triples in a relational database and builds an R-tree index for the spatial data. [69] implements an RDF storage and SPARQL query processor for mobile devices. It also stores RDF triples using a relational database and uses R-trees for spatial data indexing. Unlike existing techniques, our spatial index does not require internal modification of existing RDF systems because it is developed using only standard features.

7.3 Spatial Query Processing

In this section, we propose a lightweight and scalable spatial index, based on the hierarchical spatial data structure, for big data. We first explain how we develop the spatial index on top of HBase without modifying its internal implementation. Next, we extend our index for spatial data stored as a graph structure.

7.3.1 Overview

A spatial object basically includes its geometry and can have any additional information about the object, such as its name, address and phone number. In terms of the geometry, our spatial index supports most of generally used geometries including points, lines, rectangles, curves and polygons. Given a spatial object to be stored and indexed by our spatial index, we first calculate a set of minimum bounding boxes (i.e., geohash codes), called *minimum geohash set*, which fully cover the geometry of the spatial object. To prevent generating too many fine-grained bounding boxes to cover the geometry and thus increasing the overhead of managing the spatial object, we set the maximum number of bounding boxes for each geometry to 10 in the first prototype of our spatial index. The maximum number of bounding boxes for each geometry can be configured for different applications. Also, all the geohash codes included in a minimum geohash set have the same length and thus represent the same precision.

Definition 22 (Minimum geohash set) Given a spatial object SO with its geometry SO_G, the minimum geohash set of SO is a set of geohash codes, denoted by $minGeohash(SO) = \{geohash_1, geohash_2, \ldots, geohash_l\}$, which fully cover SO_G while minimizing the size of bounding boxes represented by the geohash codes. l is equal to or less than θ_{max}, which defines the maximum number of geohash codes for each spatial object. The minimum geohash set is defined similarly for a spatial query with its search geometry.

Similar to other indexing techniques such as R-trees, the query processing based on our spatial index basically consists of two main steps: *filter* step and *refinement* step. Given a spatial query Q, in the filter step, we find candidate spatial objects, which may satisfy the

query condition of Q, by pruning non-qualifying spatial objects. In the refinement step, we examine each candidate spatial object to determine whether the object is actually satisfying the query condition of Q. We define the *precision* of query processing for Q as the ratio of actual spatial objects satisfying the query condition of Q to all evaluated candidate spatial objects.

7.3.2 Distributed Storage Systems

To develop our spatial index on top of HBase, we propose to utilize HBase row keys to indicate the geohash codes for stored spatial objects. Specifically, given a spatial object SO to be stored and indexed by our spatial index, for each geohash code in its minimum geohash set $minGeohash(SO)$, we store the spatial object in the HBase row having the geohash code as its row key. We use an uniquely assigned identifier for the object as its column name (qualifier). We allow replication of spatial objects in multiple HBase rows for efficient processing of spatial queries as we will explain below. For example, if the minimum geohash set of a spatial object is {"dn5bpsby", "dn5bpsbv"}, we store the spatial object in two HBase rows whose keys are "dn5bpsby" and "dn5bpsbv". Note that our replication of spatial objects is not related to the data block replication of underlying HDFS for its fault-tolerance.

According to the definition of the geohash, longer geohash codes will be generated for smaller geometries. If there are many spatial objects associated with a tiny geometry, a huge number of HBase rows having a long row key may be created to store the objects and each row will likely include only a few spatial objects. Since too many HBase rows can aggravate the performance of our spatial query processing, we need to control the number of HBase rows. To limit the number of HBase rows, we utilize the hierarchical feature of the geohash codes. By setting the maximum length of geohash codes (i.e., length of HBase row keys), we can store those spatial objects associated with a tiny geometry in HBase rows representing a bigger rectangle and thus reduce the number of HBase rows. Algorithm 5 shows the pseudocode of our storing and indexing steps for a spatial object.

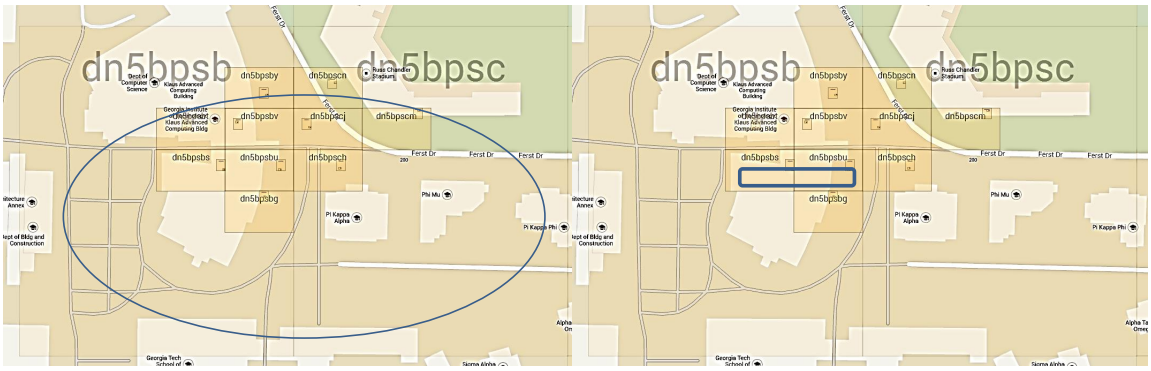
To execute spatial queries for the stored and indexed spatial objects in HBase, we utilize

Algorithm 5 Storing and indexing a spatial object

```
Input: a spatial object SO, m (the maximum length of HBase row keys)
minGeohash_SO = calculateMinimumGeohashSet(SO)
for each geohash code g in minGeohash_SO do
    if g.length > m then
        g = first m characters of g
    end if
    store SO in the HBase row whose row key is g
end for
```

the properties of the geohash codes to find only relevant HBase rows and thus reduce the search space considerably. Let us assume that a spatial query Q with its search geometry Q_G is given. We first calculate the minimum geohash set of Q, which fully covers Q_G. If the query is *containing(search geometry)*, we select only those HBase rows whose row key is a prefix of one of the geohash codes in the minimum geohash set. This is because those spatial objects that contain the search geometry should have at least the same or larger rectangles than the search geometry. As we explained above, a geohash code representing a rectangle is a prefix of those geohash codes representing the sub-rectangles of the rectangle. Therefore, we can efficiently select candidate HBase rows that may store spatial objects containing the search geometry, using the prefix match. Specifically, to find candidate HBase rows, we scan all possible prefixes for each geohash code in the minimum geohash set. For example, for a geohash code "dn5b" included in the minimum geohash set, we scan for key "d", "dn", "dn5" and "dn5b". Finally, for each candidate HBase row, we read all spatial objects stored in the row and return those spatial objects that *actually* contain the search geometry. For example, for a spatial object with its ellipsoidal geometry as shown in Fig. 50(a), we store the spatial object in two HBase rows whose row keys are "dn5bpsb" and "dn5bpsc". For a *containing* query with its rectangular search geometry as shown in Fig. 50(b), its minimum geohash set is {"dn5bpsbs", "dn5bpsbu"} and thus we select the HBase row whose key is "dn5bpsb" as a candidate row because its row key is a prefix of the geohash codes.

If the query is *containedIn(search geometry)*, an intuitive approach is to select only those HBase rows whose row key includes one of the geohash codes, included in the minimum geohash set, as its prefix because *containedIn* is the converse of *containing*. However, we need to take into account that we set the maximum length of geohash codes to prevent generating too many small HBase rows. For example, let us assume that the minimum



(a) spatial object (ellipse) (b) search geometry (rectangle)

Figure 50: Query Processing Example (*containing*)

geohash set of a spatial object is {"dn5bpsby"} and the spatial object is stored in a HBase row whose row key is "dn5bp" because the maximum length of geohash codes is 5. Also, assume that a *containedIn(search geometry)* query in which the minimum geohash set of the search geometry is {"dn5bpsb"} is given and the search geometry actually contains the spatial object. Based on the intuitive approach, we cannot select the HBase row "dn5bp" because "dn5bp" does not include "dn5bpsb" as its prefix. To tackle this problem, we also apply the maximum length to the geohash codes included in the minimum geohash set of the spatial query (from "dn5bpsb" to "dn5bp" in the previous example) and then use the intuitive approach. When we select candidate HBase rows whose row key includes one of the geohash codes, included in the minimum geohash set of the spatial query, as its prefix, we utilize a range scan of HBase for each geohash code. Specifically, for each geohash code included in the minimum geohas set, we execute a range scan whose start row is the geohash code and end row is the lexicographically next geohash code, having the same length, to access all HBase rows whose row key has the geohash code as its prefix. For example, for a geohash code "dn5b", we execute a range scan from "dn5b" to "dn5c". For each selected HBase row, we read the stored spatial objects in the row and return those spatial objects that are *actually* contained in the search geometry.

If the query is *intersects(search geometry)*, we consider both prefix cases when we select candidate HBase rows. This is because, if there is any intersecting region between the search geometry and the geometry of a spatial object, both geometries should have a rectangle(s)

Algorithm 6 Spatial Query Processing

```
Input: a spatial query Q, m (the maximum length of HBase row keys)
Output: a set of spatial objects satisfying Q
SO_Q = ∅ // a set of spatial objects satisfying Q
minGeohash_Q = calculateMinimumGeohashSet(Q) // if Q is withinDistance, use the extended geometry
for each geohash code g in minGeohash_Q do
    if g.length > m then
        g = first m characters of g
    end if
    if Q is containing or intersects or withinDistance then
        select those HBase rows whose row key is a prefix of g
        read spatial objects stored in the selected HBase rows
        add those spatial objects satisfying Q into SO_Q
    end if
    if Q is containedIn or intersects or withinDistance then
        run a range scan from g to g' where g' is the lexicographically next geohash code from g
        read spatial objects stored in the scanned HBase rows
        add those spatial objects satisfying Q into SO_Q
    end if
end for
return SO_Q
```

(i.e., geohash code) that includes the intersecting region and any two different rectangles including the same region should have their hierarchy (i.e., one is the sub-rectangle of the other) according to the definition of the geohash codes. Since we do not know which geometry has a bigger rectangle covering the intersecting region until we evaluate the spatial object, we select those HBase rows, as candidate rows, whose row key is a prefix of one of the geohash codes included in the minimum geohash set of the spatial query *or* includes one of the geohash codes as its prefix. For each selected HBase row, we read the stored spatial objects in the row and return those spatial objects that are *actually* intersecting with the search geometry.

For a *withinDistance(search geometry, distance)* query, we first calculate the minimum geohash set, which covers the extended geometry computed by adding the *distance* to the search geometry. Then, similar to the *intersects* query processing, we select those HBase rows, as candidate rows, whose row key is a prefix of one of the geohash codes included in the minimum geohash set *or* includes one of the geohash codes as its prefix. For each selected HBase row, we read the stored spatial objects in the row and return those spatial objects that are *actually* within the *distance* from the the search geometry. Algorithm 6 shows the pseudocode of our spatial query processing.

In addition to HBase, our index can also be used to improve the performance of Hadoop

MapReduce programs handling spatial objects. Most Hadoop MapReduce programs basically read and evaluate all the records stored in their input HDFS paths via their map function because they have no information about the stored records before reading them. With our spatial index on top of HDFS where we use geohash codes of spatial objects as HDFS file names, Hadoop MapReduce programs can read and evaluate only relevant files by simply implementing and setting their *PathFilter*, which describes a set of files they want to access and thus considerably reduce the input record size to be read and evaluated, without any help from the external and complicated indices. For example, if a Hadoop MapReduce program wants to analyze only those records included in a specific city and the geohash code of the city is "dn5bp", the Hadoop job can reduce the input record size by reading only relevant HDFS files whose file name has "dn5bp" as its prefix.

7.3.3 Graph Models

We extend our spatial index for spatial objects represented as a graph model. To develop our spatial index on top of RDF, for each vertex representing a spatial object, we add an RDF triple (edge) storing the geohash code of the spatial object. Specifically, for each spatial object with its geometry, we first calculate the minimum geohash set, which fully covers the geometry of the spatial object. For each geohash code included in the minimum geohash set, we add an edge, representing the geohash code, to the vertex denoting the spatial object. For example, if a calculated geohash code for a spatial object denoted as <http://cc.gatech.edu/disl/Object1> is "dn5bpsby", we add a triple in which its subject, predicate and object are <http://cc.gatech.edu/disl/Object1>, <http://cc.gatech.edu/disl/geohash> and "dn5bpsby" respectively. <http://cc.gatech.edu /disl/geohash> is a predicate representing a geohash relationship from a spatial object to a geohash code. Fig. 51 is an example RDF graph that shows how geohash codes are added for three spatial objects. Algorithm 7 shows the pseudocode of our storing and indexing steps on top of RDF for a spatial object.

In order to execute spatial queries for the indexed spatial objects based on the RDF graph model, we utilize SPARQL, which is a standard query language for RDF. Specifically,

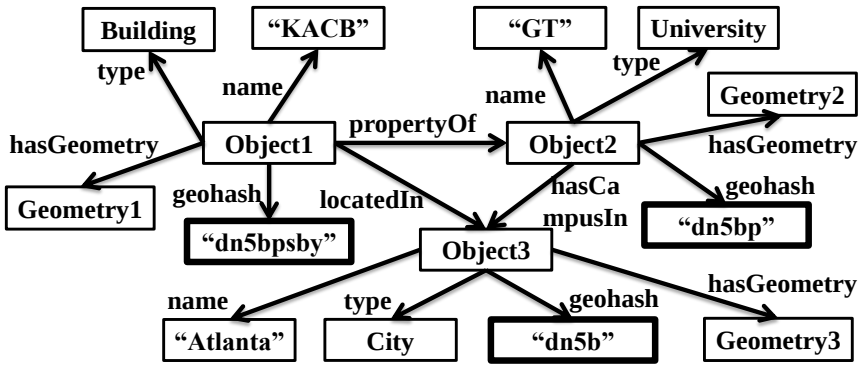


Figure 51: RDF Graph with Geohash Codes

Algorithm 7 Storing and indexing a spatial object (RDF)

Input: a spatial object SO
1: $vertex_{SO} = createSpatialObjectVertex(SO)$
2: $minGeohash_{SO} = calculateMinimumGeohashSet(SO)$
3: **for** each geohash code g in $minGeohash_{SO}$ **do**
4: store a triple ($< vertex_{SO} > < geohash >$ "g")
5: **end for**
6: store triples representing other information of SO

to represent our spatial queries (*containing*, *containedIn*, *intersects* and *withinDistance*) in SPARQL, we adopt the syntax of GeoSPARQL [39] and thus include the spatial queries in a SPARQL filter. We call a filter including any spatial query a *spatial filter*. For example, the below example query has a *containing* spatial filter.

```
PREFIX gt: <http://cc.gatech.edu/disl/>
SELECT ?so
WHERE {
    ?so gt:hasGeometry ?geometry .
    FILTER(gt:containing(?geometry,
        "Point(-83.4 34.3)"^^gt:wktLiteral)) }
```

If a SPARQL query including any spatial filter is given, we rewrite the spatial filter using a set of prefix filters based on the geohash codes and thus execute the query using only standard features of SPARQL. Specifically, similar to our spatial query processing on top of HBase, we first calculate the minimum geohash set, which fully covers the search geometry given in the spatial filter. If the query is *containedIn*, we need to find those candidate spatial objects whose geohash code includes one of the calculated geohash codes as its prefix. To find

such spatial objects, we utilize the regular expression prefix filter of SPARQL. For example, if the calculated geohash codes of the search geometry are "dn5bpsb" and "dn5bpsc", we rewrite the spatial filter into a filter based on the prefix matching as follows:

$$FILTER(regex(?geohash, \text{“}\hat{}dn5bpsb\text{”})||regex(?geohash, \text{“}\hat{}dn5bpsc\text{”}))$$

If the query is *containing*, we need to find those candidate spatial objects whose geohash code is a prefix of one of the geohash codes included in the minimum geohash set. To find such spatial objects, we utilize the exact match filter of SPARQL for each possible prefix of the calculted geohash codes. For example, if the calculated geohash code of the search geometry is "dn5b", we rewrite the spatial filter into a filter based on the exact matching as follows:

$$FILTER(?geohash = \text{“}d\text{”}||?geohash = \text{“}dn\text{”}||?geohash = \text{“}dn5\text{”}||?geohash = \text{“}dn5b\text{”})$$

If the query is *intersects* or *withinDistance*, we rewrite the spatial filter using both prefix filter and exact filter of SPARQL. This is because we need to find those candidate spatial objects whose geohash code is a prefix of one of the calculated geohash codes *or* includes one of the calculated geohash codes as its prefix. Recall that we get *candidate* spatial objects by running the rewritten SPARQL query and thus we need a final step, which finds those spatial objects that *actually* satisfy the given query among the candidate spatial objects.

Even though the standard SPARQL includes the regular expression prefix matching, its implementation in the RDF and SPARQL systems may be inefficient if no proper index is constructed for processing the prefix matching. To tackle this inefficiency, we propose an alternative approach that can also be implemented on top of RDF without any modification of the standard model and existing RDF systems. Its basic idea is, for each geohash code of a spatial object, to add multiple edges representing different precisions (lengths) of the geohash code and utilize the exact matching for spatial query processing. The primary motivation of this approach is that most RDF systems efficiently support the exact match filter using a set of indices and adding some more edges (triples) has little effect on their query processing performance, thanks to the indices. Specifically, let us first assume that search geometries usually have their geohash codes having length of from l to $l + k$

characters. For each calculated geohash code of a spatial object, we add multiple edges representing different precisions (lengths), from l characters to $l+k$ characters by removing characters from the end of the geohash code. In other words, if the length of the calculated geohash code is equal to or longer than $l+k$, $k+1$ edges representing different precisions will be added. For spatial query processing, if the length of the calculated geohash codes of a search geometry is between l and $l+k$ (inclusive), we rewrite the spatial filter using the exact filter, instead of the prefix filter. Otherwise, we rewrite the spatial filter using the prefix filter of SPARQL. For example, if a calculated geohash code for a spatial object denoted as <http://cc.gatech.edu/disl/object1> is "dn5bpsby" and search geometries usually have their geohash codes having length of from 3 to 5 characters, we add three edges representing geohash codes "dn5", "dn5b" and "dn5bp". Given a *containedIn* query in which the calculated geohash codes of the search geometry are "dn5bn" and "dn5bp", we rewrite the spatial filter into a filter based on the exact matching as follows:

$$FILTER(?geohash = \text{“}dn5bn\text{”} || ?geohash = \text{“}dn5bp\text{”})$$

Our spatial index on top of RDF does not require any internal modification of RDF and SPARQL, which are standards. That means we can directly utilize any existing RDF and SPARQL systems. The only requirement for spatial query processing is the SPARQL rewriter, which rewrites spatial filters into standard SPARQL filters.

7.4 Experimental Evaluation

In this section, we report the experimental evaluation of our spatial index on top of HBase and RDF. We first present spatial query processing performance using our index on top of HBase. Next we show the experimental results, including the comparison of different SPARQL filters, on spatial query processing using our index on top of RDF. We also compare the pruning power of our spatial index with that of an R-tree-based index.

7.4.1 Experimental Setup and Datasets

For evaluation of our spatial index on top of HBase, we use HBase (Version 0.96) and Hadoop (Version 1.0.4) running on Java 1.6.0, installed on a cluster of 11 physical machines

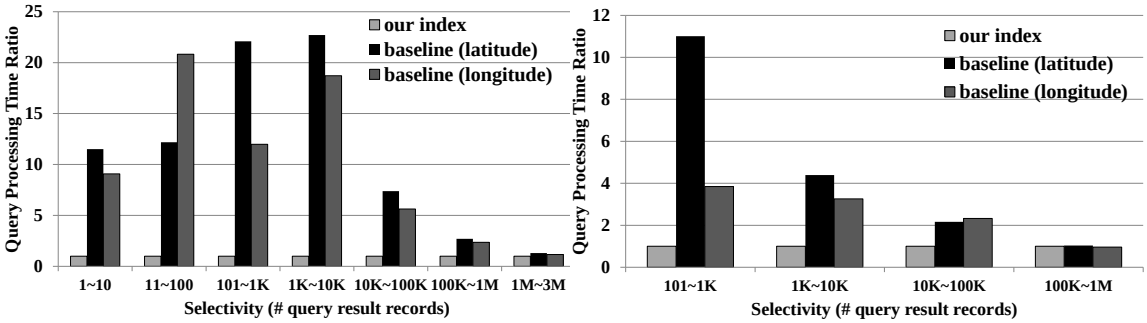
(one master machine) on Emulab [116]: each has 12GB RAM, one 2.4 GHz 64-bit quad core Xeon E5530 processor and two 7200 rpm SATA disks (500GB and 250GB). We run HBase RegionServers on the same machines as DataNodes and a ZooKeeper ensemble of 3 machines. For each setting and each query, our spatial query processing time indicates the fastest time after running five cold runs to remove any possible bias posed by OS and/or network activity. For evaluation of our spatial index on top of RDF, we use DB2RDF of DB2 10.5 Express-C installed on an Emulab machine having the same specifications like the above ones.

We use GeoLife GPS Trajectories (GeoLife in short) [125] and San Francisco taxi cab traces (SFTaxi in short) [95] for our experiments. GeoLife contains 24,876,977 GPS point records (17,621 trajectories), gathered by 182 users in a period of over five years (from April 2007 to August 2012), with a total distance of about 1.3 million kilometers and a total duration of about 50,000 hours. SFTaxi contains 11,219,955 GPS point records, collected over 30 days, of approximately 500 taxi cabs in San Francisco, USA. For evaluation of our spatial index on top of RDF, we convert the two datasets into RDF-formatted files (N-Triples).

7.4.2 Distributed Storage Systems

We first present spatial query processing performance using our index on top of HBase running on HDFS. As our baseline approach, we store the spatial objects using their latitude (or longitude) as a row key of HBase (i.e., one dimensional index). We choose this approach as our baseline because it can be also implemented without modifying HBase and, similar to our spatial index, HBase range scans can be utilized for fair comparisons. For example, given a *containedIn* query, we use the leftmost and rightmost latitudes (or longitudes) of the query geometry as the start and end row keys of a HBase range scan respectively.

We implement a Hadoop MapReduce job to efficiently store the spatial objects in HBase. Also, we represent each geohash code as a binary array, instead of a string, to efficiently handle geohash codes. By default, we empirically choose 40 bits as the maximum length of geohash codes because we think that value strikes a balance between the number of rows



(a) *withinDistance* queries (b) *containedIn* queries

Figure 52: Query Processing Time

and the number of columns of each row. We will show the effects of different maximum lengths of geohash codes in this section. 2,608,848, 4,744,257 and 4,886,185 HBase rows are generated to store the spatial objects using our index, the latitude-based baseline approach and longitude-based baseline approach respectively.

In this chapter, we report the results of *withinDistance* and *containedIn* queries. We do not include the results of *containing* and *intersects* queries because they have similar query processing results with *withinDistance* and *containedIn* queries. We generate 300 *withinDistance* queries by randomly selecting a point in the datasets and using a distance of 10m, 100m or 1km. This generation process guarantees that we get at least one point record as the output of each query execution. We also generate 100 *containedIn* queries by randomly selecting two points in the datasets and using them as the lower-left and upper-right points of a rectangle.

For brevity, we first categorize the queries based on their selectivity and then compare our query processing performance with that of the baseline approach using the ratio of their query processing times where we set our query processing time to 1, as shown in Fig. 52. The query processing with our spatial index is more than one order of magnitude faster than both the latitude-based and longitude-based baseline approaches, on average, for those *withinDistance* queries that select less than 10,000 records, as shown in Fig. 52(a). As we decrease the selectivity of queries, the performance gain of our spatial index also drops because retrieving a large number of rows for query evaluation is inevitable. However, the query processing with our spatial index is still 30% faster than the latitude-based baseline

approach, on average, for those *withinDistance* queries that select more than 1 million records. For *containedIn* queries, even though our query processing is still more than one order of magnitude faster than the latitude-based baseline approach for queries having high selectivity as shown in Fig. 52(b), its performance gain is generally smaller than that for *withinDistance* queries. This is primarily because *containedIn* queries usually cover a wider region than *withinDistance* queries and thus the pruning power of the baseline approaches is higher for *containedIn* queries. Specifically, the average precisions (i.e., the ratio of true positives to all evaluated candidate spatial objects) of the latitude-based baseline approach are 8% and 12% for *withinDistance* queries and *containedIn* queries respectively.

Table 20: Query Processing Results (*withinDistance*)

query	**# result records**		**our index**	**baseline (lat)**	**baseline (long)**
Q1	3	time(sec)	0.004	0.188	0.070
		# cand. records	18	6060	1074
		# accessed rows	4	582	380
Q2	20	time(sec)	0.005	0.107	0.085
		# cand. records	94	2855	18429
		# accessed rows	2	310	584
Q3	271	time(sec)	0.028	1.161	0.104
		# cand. records	2896	233511	21027
		# accessed rows	4	2280	603
Q4	3370	time(sec)	0.054	2.222	1.257
		# cand. records	9618	478920	276189
		# accessed rows	204	10462	7619
Q5	107K	time(sec)	1.511	5.427	6.509
		# cand. records	294K	1003K	1066K
		# accessed rows	9K	57K	171K
Q6	1020K	time(sec)	13.803	21.348	25.275
		# cand. records	2640K	3999K	5396K
		# accessed rows	14K	191K	78K

To provide more detailed analysis of the query processing results, we include specific results of some selected queries in Table 20. The query processing times are basically related to the number of evaluated records (i.e., candidate records). This is because more candidate records for the same query mean that the query processor wastes more time for evaluating false positive records. The results clearly show much higher pruning power of our spatial index, compared to the baseline approaches, which consider only one dimension. For Q4, for example, to find 3,370 records satisfying the query condition, our spatial index-based approach evaluates 9,618 records (i.e., its precision is 35%). On the other hand, the latitude-based baseline approach evaluates 478,920 records and thus its precision is less than 1% for

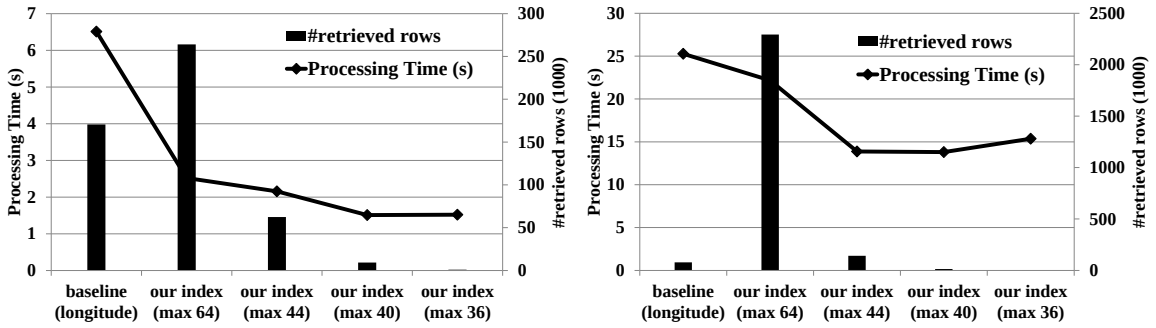
the query.

Table 21: Breakdown of Query Processing Results

seconds		**Our index**	**baseline (lat)**	**baseline (long)**
Q1	Index Access	0.004	0.171	0.068
	Evaluation	0.000	0.017	0.002
Q2	Index Access	0.005	0.100	0.036
	Evaluation	0.000	0.007	0.049
Q3	Index Access	0.021	0.493	0.049
	Evaluation	0.007	0.668	0.055
Q4	Index Access	0.028	0.889	0.530
	Evaluation	0.026	1.333	0.727
Q5	Index Access	0.672	2.552	3.560
	Evaluation	0.839	2.875	2.949
Q6	Index Access	6.277	9.607	11.160
	Evaluation	7.526	11.741	14.115

Table 21 shows the index access time (i.e., range scan time) and the evaluation time of the candidate records for the six queries. Our spatial index-based approach has the fastest evaluation time because it evaluates much smaller number of records than the baseline approaches as shown in Table 20. Our approach also has the fastest index access time because it reads the smallest number of rows for each query. It is interesting to note that the index access time is not linearly proportional to the number of accessed rows. For example, even though the longitude-based baseline approach reads about 150 times more rows than our spatial index-based approach for Q3, its index access time is only 2 times slower. This is primarily because, for each query, our spatial index-based query processing usually consists of multiple (mostly from 2 to 4) range scans while the baseline approach always uses one range scan. Furthermore, the processing time of each range scan is usually not proportional to the number of accessed rows (or retrieved records) due to several factors including the block cache of HBase RegionServers.

Fig. 53 shows the effects of different maximum geohash lengths for query processing. We report the results of only Q5 and Q6 in Table 20 because the difference in terms of query processing time is negligible for Q1, Q2, Q3 and Q4. In other words, our spatial index provides efficient query processing performance, regardless of the maximum geohash lengths, for those queries having high selectivity. In addition to 40 bits as the maximum length of geohash codes, we store the spatial objects in HBase using 64 bits (which keep full geohash codes), 44 bits and 36 bits. 27,973,524, 7,620,735 and 754,860 HBase rows



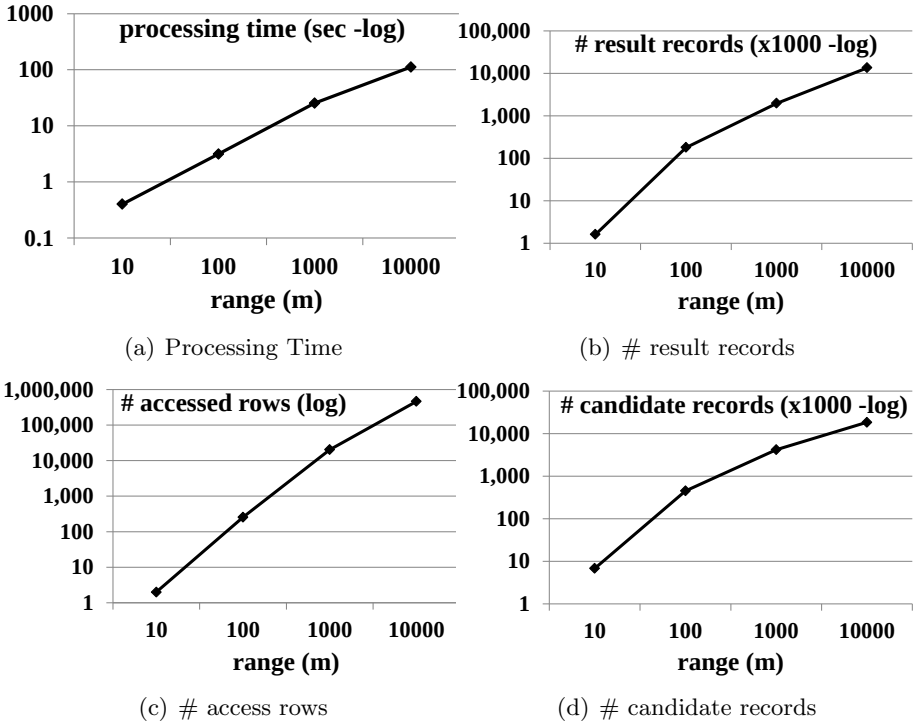
(a) Q5 (b) Q6

Figure 53: Effects of Different Maximum Lengths

are generated for 64, 44 and 36 bits respectively. For each of both queries, the number of evaluated candidate records is the same regardless of the maximum geohash lengths. Query processing performance is improved as we reduce the maximum geohash length to a certain point (40 bits for both queries). This is because our spatial index using a shorter maximum geohash length retrieves a smaller number of rows as shown in Fig. 53. However, at a certain point, reducing the maximum geohash length does not improve the query processing performance any more (or even aggravate the performance) because the query processor should read and process very wide rows having a lot of columns for query evaluation. It also requires more main memory for query processing. Note that, for Q5, our spatial index with full geohash codes (64 bits) provides more than two times better query processing performance than the baseline approach because, even though it retrieves more rows than the baseline approach as shown in Fig. 53(a), its number of evaluated candidate records is only 30% of that of the baseline approach.

Fig. 54 shows the query processing results using different distances for the same query point of a *withinDistance* query. The query processing time understandably increases as we enlarge the query region because more HBase rows are accessed and thus more candidate records are evaluated for query processing.

Fig. 55 shows the insertion time of new 100 records for different dataset sizes (i.e., the number of already stored records). For this experiment, we insert the new records from the master machine. Since our spatial index-based approach does not create any expensive data structure unlike existing techniques including R-tree-based approaches, the insertion



(a) Processing Time

(b) # result records

(c) # access rows

(d) # candidate records

Figure 54: Effects of Different Distances

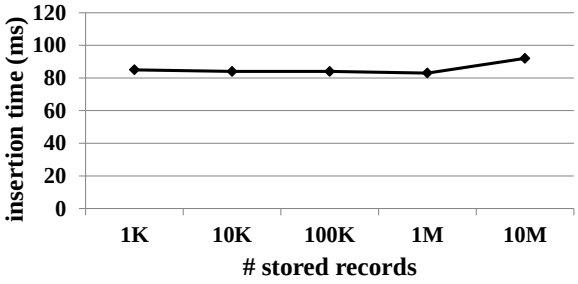


Figure 55: Insertion Time

time is almost constant regardless of the dataset size.

7.4.3 Graph Models

In this section, we present spatial query processing performance using our index on top of RDF. We implement our index in two different ways, using the prefix matching and exact matching, to compare their performance in the RDF system. For our index using the prefix matching, we store full geohash codes in the RDF system. As our baseline approach, we utilize SPARQL comparison filters, in which we set the lower-left and upper-right points of a query bounding box, because we think this approach is the most intuitive approach

for processing spatial queries. For example, given a *withinDistance* query, we calculate a minimum bounding box, which fully covers its query region and then create a SPARQL query with a comparison filter that sets the lower-left and upper-right points of the bounding box. For this set of experiments, we run the very same sets of queries, which are used for the evaluation of our index on top of HBase.

Table 22: SPARQL Query Processing Time Ratio

selectivity	**our index (exact match)**	**our index (prefix match)**	**baseline**
***withinDistance* queries**			
1-10	1	1.08	5251.80
11-100	1	1.02	3913.76
101-1K	1	1.16	1568.79
1K-10K	1	1.23	298.53
10K-100K	1	1.23	24.03
100K-1M	1	1.28	3.71
1M-3M	1	1.28	1.09
***containedIn* queries**			
101-1K	1	1.05	1061.97
1K-10K	1	1.19	64.24
10K-100K	1	1.25	13.65
100K-1M	1	1.31	1.69

Table 22 shows the average query processing time ratios of three different approaches for different selectivity levels. For those queries that select less than 1,000 records, the query processing with our spatial index is more than three orders of magnitude faster than the baseline approach on average. This result shows the overhead of join processing because the baseline approach touches two different predicates (i.e., latitude and longitude) in the comparison filter. We believe that the join processing is aggravated since we use a free edition of DB2, which limits the usage of main memory and the number of CPU cores. On the other hand, since our approach handles only one predicate (i.e., there is no join processing for our index), we can process spatial queries efficiently regardless of such limitations. The results also validate our claim that using the exact filter would be more efficient than using the prefix filter due to the efficient support of the exact filter in the RDF system. It is also interesting to note that, even though the baseline approach has 100% precision (i.e., no false positive record) for *containedIn* queries due to the rectangular query geometry, it is consistently slower than our approach primarily because of join processing.

Table 23 presents specific SPARQL query processing results of the same queries in

Table 23: SPARQL Query Processing Results (*withinDistance*)

		Our index			baseline	
query	**#records**	**time(s) exact**	**time(s) prefix**	**#cand. records**	**time(s)**	**#cand. records**
Q1	3	0.041	0.044	18	65.991	3
Q2	20	0.045	0.045	94	69.287	28
Q3	271	0.139	0.153	2896	69.220	350
Q4	3370	0.238	0.378	9618	69.456	3454
Q5	107K	7.611	10.746	294K	69.442	121K
Q6	1020K	75.254	100.399	2640K	101.450	1261K

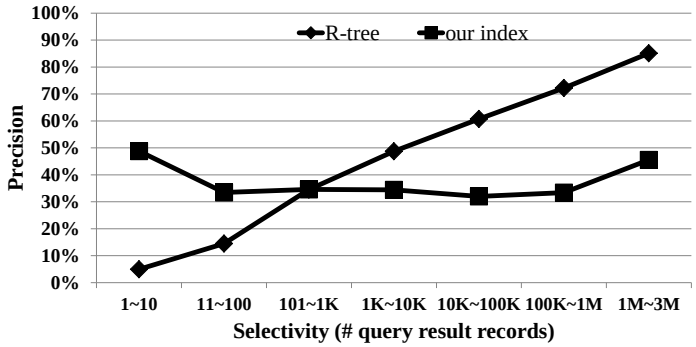


Figure 56: Precision Comparison (*withinDistance*)

Table 20. For those queries having very high selectivity (Q1, Q2 and Q3), our approach using the prefix filter is comparable to that using the exact filter. This result validates that the prefix filter is efficiently supported by the RDF system. Nevertheless, the exact match-based approach is about 30% faster than the prefix match-based approach for the other queries having low selectivity. The baseline approach is much slower even though it has much higher precision.

7.4.4 Comparison with R-tree

Finally, we compare the pruning power of our spatial index with that of an R-tree-based index. We use an open source R-tree implementation [13] for this evaluation. We want to emphasize that the focus of this chapter is on the scalable and lightweight spatial index, which can be easily applied to existing systems without modifying their internal implementation. Outperforming the pruning power of R-tree-based indices is not the purpose of this chapter because R-tree-based indices maintain expensive data structures and mostly require internal and complicated modification of the storage systems. Nevertheless, the precision results in Fig. 56 show that our index has one order of magnitude higher precision than

the R-tree-based index for those queries having very high selectivity (selecting less than 10 records). Our spatial index demonstrates relatively consistent precision for different selectivity levels while the R-tree-based index has higher precision for less selective queries.

7.5 Conclusion

In this chapter we have proposed efficient and scalable spatial indexing techniques for big data stored in distributed storage systems or graph models. Based on a hierarchical spatial data structure, called geohash, we have presented how we develop a lightweight spatial index for big data stored in a distributed file system, especially on top of HBase. In addition, we have described how we extend our spatial index for graph data, especially on top of RDF. Our spatial index has several advantages. First, it can be easily applied to existing storage systems or graph models without modifying their internal implementation. Second, it provides an efficient pruning technique that can find only relevant spatial objects based on prefix matching. Third, it supports customizable control of index size for different applications. Our experimental results show the efficiency and effectiveness of our spatial indexing techniques.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

With continued advances in computing and information technology, digital data have grown at an astonishing rate in terms of volume, variety, and velocity. Such big data have huge potential to reveal hidden insights and promote innovation in many business, science, and engineering domains. An important technical challenge faced by many big data systems and applications is how to build efficient big data processing systems and applications that can scale to the rapid growth of digital data in the 21st century. To address this challenge, this dissertation is dedicated to the development of architectures and optimization techniques for scaling big data processing systems, especially in the era of cloud computing. In this chapter, we first summarize the main contributions of this dissertation and then discuss our future research directions.

8.1 Summary

In summary, this dissertation makes three unique contributions. First, it introduces a suite of graph partitioning algorithms that can run much faster than existing data distribution methods and inherently scale to the growth of big data. The main idea of these approaches is to partition a big graph by preserving the core computational data structure as much as possible to maximize intra-server computation and minimize inter-server communication. Based on this main idea, we present a new distributed RDF system, called Shape, to improve the performance of distributed RDF query processing. Shape is equipped with a scalable partitioning technique and an efficient distributed query processing technique. The experimental evaluation on large graphs with hundreds of millions of vertices and billions of edges has shown that Shape, which can scale to large graphs with varying sizes and complexity, is more efficient than existing distributed RDF systems. We also propose a distributed graph partitioning framework called VB-Partitioner, which supports efficient graph query processing for various graph data characteristics and query workloads.

Equipped with three different grouping techniques, VB-Partitioner significantly outperforms the popular random block-based graph partitioning in terms of query latency.

In addition, this dissertation proposes a distributed iterative graph computation framework called GraphMap, which effectively utilizes secondary storage to maximize access locality and speed up distributed iterative graph computations. To address the poor scalability of existing distributed graph systems, we distinguish read-only graph data from mutable graph data and store the read-only data on disk. To maximize sequential disk access and minimize random disk access, we also propose locality-optimized data placement based on a two-level graph partitioning algorithm. Furthermore, we develop locality-based optimization, which dynamically chooses between sequential disk access and random disk access based on the computation loads of each iteration for each worker machine. The framework not only considerably reduces memory requirements for iterative graph algorithms but also significantly improves the performance of iterative graph computations.

Last but not the least, this dissertation establishes a suite of optimization techniques for scalable spatial data processing along with three orthogonal dimensions. First, we develop a road network-aware spatial alarm processing system called RoadAlarm. RoadAlarm offers a suite of alarm processing and optimization techniques that minimize the amount of wakeups at mobile clients to save energy while reducing the amount of unnecessary alarm checks at the server to improve the server performance and the accuracy of alarm evaluations. Second, we propose a framework to predict the location of each tweet on Twitter. We build probabilistic models for locations using data from Foursquare, which is another social network specialized in locations, instead of noisy data from Twitter. To increase the accuracy of prediction, we evaluate various ranking methods, smoothing techniques, and language models. Third, we introduce an efficient and lightweight spatial index for big data stored in distributed storage systems. Based on a hierarchical spatial data structure, the index can be easily applied to existing storage systems without modifying their internal implementation.

8.2 Future Work

There are many interesting open research problems for scalable big data processing from various perspectives. In the context of big graph processing, our future research interests include improving GraphMap for better performance of iterative graph computations, summarizing graphs for more efficient graph query processing, and ultimately developing a unified system that efficiently and effectively supports both graph query processing and iterative graph computations. We are also interested in elastic cloud computing technologies for big data processing, including cost-efficient resource management for Platform-as-a-Service (PaaS) systems based on operating system-level virtualization (containers). We highlight some of them below.

8.2.1 Scalable Systems for Big Graph Data Analytics

Several directions of our ongoing research fall into this category. First of all, to further tackle the scalability challenges of iterative graph computations, we will focus on utilizing secondary storage even for storing mutable graph data in addition to read-only data. Naive utilization of disks for storing mutable graph data would severely aggravate the performance of iterative graph computations because it would require a huge number of random disk accesses. We look forward to devising systematic approaches to storing mutable graph data by taking into account data access locality while ensuring competitive performance compared to distributed memory-based systems. In addition, we will extend GraphMap to support other storage systems such as GraphChi and X-Stream on each compute node and to include more efficient and lightweight partitioning techniques. Second, we will work on graph summarization techniques that efficiently execute complex graph queries including many graph patterns. In most graph systems, running complex graph queries is very costly, mostly because of many internal joins. Our basic idea would add more edges that summarize complex graph patterns to the original graph. Then, given a complex query, we will convert the query into a simpler query using the summarized edges. Last but not the least, our ultimate research goal in the context of big graph processing is to develop a unified distributed system that supports both graph query processing and iterative graph

computations efficiently.

8.2.2 Cost-Efficient Resource Management in Cloud Computing

The Platform-as-a-Service (PaaS) cloud computing service model is in the limelight of both industry and academia as the operating system in the cloud. By hiding the complexities of IaaS, PaaS makes it easier to write cloud applications not only for application developers but also small and medium-sized businesses that rarely have enough IT personnel for managing an on-premise or leased cloud-computing infrastructure. Most PaaS providers offer free trials that entice customers and ultimately make them pay for the PaaS services. Unfortunately, most free-trial users start using PaaS out of curiosity, and to make matters worse, they never access their applications after creating or running the applications. Therefore, to serve those users who are very unlikely to pay for the PaaS services, PaaS providers waste precious resources (and thus money). To tackle this challenge, we will work on designing a new framework that dynamically adapts PaaS to optimize the use of its resources for various service level agreements. Since many PaaS systems are using operating system-level virtualization (containers) to run application instances, we will focus on how to manage containers with minimal overhead. This framework will also include prediction models to infer required computing resources by analyzing data and computation characteristics. In addition, we will develop container allocation models to optimally assign containers based on the predicted resource requirements.

REFERENCES

[1] "About FacetedDBLP," *http://dblp.l3s.de/dblp++.php*.

[2] "About foursquare." https://foursquare.com/about/.

[3] "Apache Giraph," *http://giraph.apache.org/*.

[4] "Apache Hama," *https://hama.apache.org/*.

[5] "BTC 2012," *http://km.aifb.kit.edu/projects/btc-2012/*.

[6] "Chaco: Software for Partitioning Graphs," *http://www.sandia.gov/ bahendr/chaco.html*.

[7] "DBpedia 3.8 Downloads," *http://wiki.dbpedia.org/Downloads38*.

[8] "Geo Developer Guidelines," *https://dev.twitter.com/terms/geo-developer-guidelines*.

[9] "Geohash," *http://geohash.org/*.

[10] "GPoSTTL," *http://gposttl.sourceforge.net/*.

[11] "GT-mobisim," *http://code.google.com/p/gt-mobisim/*.

[12] "IDC: 87% Of Connected Devices Sales By 2017 Will Be Tablets And Smartphones," *http://onforb.es/14J7vkl*.

[13] "JSI RTree Library," *http://jsi.sourceforge.net/*.

[14] "Linking Open Data," *http://www.w3.org/wiki/SweoIG /TaskForces/CommunityProjects/LinkingOpenData*.

[15] "METIS," *http://www.cs.umn.edu/~metis*.

[16] "Optimize Your Settings," *http://www.apple.com/batteries/iphone.html*.

[17] "Resource Description Framework (RDF)," *http://www.w3.org/RDF/*.

[18] "Snowball," *http://snowball.tartarus.org/*.

[19] "SPARQL Query Language," *http://www.w3.org/TR/rdf-sparql-query/*.

[20] "Twitter: A New Age for Customer Service - Forbes," *http://onforb.es/VqqTxa*.

[21] "Twitter Decahose," *http://gnip.com/twitter/decahose*.

[22] "U.S. Geological Survey," *http://www.usgs.gov/*.

[23] Akdogan, A., Demiryurek, U., Banaei-Kashani, F., and Shahabi, C., "Voronoi-Based Geospatial Query Processing with MapReduce," in *CLOUDCOM*, 2010.

[24] Amitay, E., Har'El, N., Sivan, R., and Soffer, A., "Web-a-where: geotagging web content," in *SIGIR*, 2004.

[25] Andreev, K. and Räcke, H., "Balanced graph partitioning," in *SPAA*, 2004.

[26] Backstrom, L., Huttenlocher, D., Kleinberg, J., and Lan, X., "Group Formation in Large Social Networks: Membership, Growth, and Evolution," in *KDD*, 2006.

[27] Bamba, B., Liu, L., Iyengar, A., and Yu, P. S., "Distributed processing of spatial alarms: A safe region-based approach," in *ICDCS*, 2009.

[28] Bamba, B., Liu, L., Yu, P. S., Zhang, G., and Doo, M., "Scalable processing of spatial alarms," in *HiPC*, 2008.

[29] Barbosa, L. and Feng, J., "Robust sentiment detection on Twitter from biased and noisy data," in *COLING*, 2010.

[30] Barceló, P., Libkin, L., and Reutter, J. L., "Querying graph patterns," in *PODS*, 2011.

[31] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B., "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," in *SIGMOD*, 1990.

[32] Benevenuto, F., Magno, G., Rodrigues, T., and Almeida, V., "Detecting spammers on Twitter," in *CEAS*, 2010.

[33] Boldi, P. and Vigna, S., "The WebGraph framework I: Compression techniques," in *WWW*, 2004.

[34] Bu, Y., Borkar, V., Jia, J., Carey, M. J., and Condie, T., "Pregelix: Big(Ger) Graph Analytics on a Dataflow Engine," *Proc. VLDB Endow.*, vol. 8, pp. 161–172, Oct. 2014.

[35] Chen, S. F. and Goodman, J., "An empirical study of smoothing techniques for language modeling," in *ACL*, 1996.

[36] Cheng, Z., Caverlee, J., and Lee, K., "You are where you tweet: a content-based approach to geo-locating twitter users," in *CIKM*, 2010.

[37] Cheng, Z., Caverlee, J., Lee, K., and Sui, D., "Exploring Millions of Footprints in Location Sharing Services," in *ICWSM*, 2011.

[38] Cho, H.-J. and Chung, C.-W., "An efficient and scalable approach to cnn queries in a road network," in *VLDB*, 2005.

[39] Consortium, O. G. and others, "OGC GeoSPARQL-A geographic query language for RDF data," 2012.

[40] Dijkstra, E. W., "A note on two problems in connexion with graphs.," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[41] Doo, M., Liu, L., Narasimhan, N., and Vasudevan, V., "Efficient indexing structure for scalable processing of spatial alarms," in *GIS*, 2010.

[42] Erling, O. and Mikhailov, I., "Towards web scale RDF," *Proc. SSWS*, 2008.

[43] Franke, C., Morin, S., Chebotko, A., and others, "Distributed Semantic Web Data Management in HBase and MySQL Cluster," in *IEEE CLOUD*, 2011.

[44] G. Karypis and V. Kumar, "A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm," in *PARALLEL PROCESSING FOR SCIENTIFIC COMPUTING. SIAM*, 1997.

[45] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C., "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," in *OSDI*, 2012.

[46] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I., "GraphX: Graph Processing in a Distributed Dataflow Framework," in *OSDI*, 2014.

[47] Guo, Y., Pan, Z., and Heflin, J., "LUBM: A benchmark for OWL knowledge base systems," *Web Semant.*, 2005.

[48] Guttman, A., "R-trees: A Dynamic Index Structure for Spatial Searching," in *SIGMOD*, 1984.

[49] Han, W.-S., Lee, S., Park, K., Lee, J.-H., Kim, M.-S., Kim, J., and Yu, H., "TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC," in *KDD*, 2013.

[50] Harth, A., Umbrich, J., Hogan, A., and Decker, S., "YARS2: a federated repository for querying graph structured data from the web," in *ISWC*, 2007.

[51] Hecht, B., Hong, L., Suh, B., and Chi, E. H., "Tweets from Justin Bieber's heart: the dynamics of the location field in user profiles," in *CHI*, 2011.

[52] Hendrickson, B. and Leland, R., "A multilevel algorithm for partitioning graphs," in *Supercomputing*, 1995.

[53] Hjaltason, G. R. and Samet, H., "Distance browsing in spatial databases," *ACM Trans. Database Syst.*, vol. 24, June 1999.

[54] Hu, H., Xu, J., and Lee, D. L., "A generic framework for monitoring continuous spatial queries over moving objects," in *SIGMOD*, 2005.

[55] Huang, J., Abadi, D. J., and Ren, K., "Scalable SPARQL Querying of Large RDF Graphs." PVLDB, 2011.

[56] Husain, M., McGlothlin, J., Masud, M. M., and others, "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing," *IEEE TKDE*, 2011.

[57] Ikawa, Y., Enoki, M., and Tatsubori, M., "Location inference using microblog messages," in *WWW Companion*, 2012.

[58] Iwerks, G. S., Samet, H., and Smith, K., "Continuous k-nearest neighbor queries for continuously moving points with updates," in *VLDB*, 2003.

[59] Jones, R., Kumar, R., Pang, B., and Tomkins, A., ""I know what you did last summer": query logs and user privacy," in *CIKM*, 2007.

[60] Karypis, G. and Kumar, V., "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM J. Sci. Comput.*, 1998.

[61] Karypis, G. and Kumar, V., "Analysis of multilevel graph partitioning," in *Supercomputing*, 1995.

[62] Karypis, G. and Kumar, V., "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Supercomputing*, 1996.

[63] Karypis, G. and Kumar, V., "Multilevel algorithms for multi-constraint graph partitioning," in *Supercomputing*, 1998.

[64] Kavanaugh, A., Yang, S., Sheetz, S. D., and Fox, E. A., "Microblogging in Crisis Situations: Mass Protests in Iran, Tunisia, Egypt," in *CHI Workshop*, 2011.

[65] Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., and Kalnis, P., "Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing," in *EuroSys*, 2013.

[66] Kwak, H., Lee, C., Park, H., and Moon, S., "What is Twitter, a Social Network or a News Media?," in *WWW*, 2010.

[67] Kyrola, A., Blelloch, G., and Guestrin, C., "GraphChi: large-scale graph computation on just a PC," in *OSDI*, 2012.

[68] Ladwig, G. and Harth, A., "CumulusRDF: Linked data management on nested key-value stores," in *SSWS*, 2011.

[69] Le Phuoc, D., Parreira, J. X., Reynolds, V., and Hauswirth, M., "RDF On the Go: RDF Storage and Query Processor for Mobile Devices.," in *ISWC*, 2010.

[70] Lee, K. and Liu, L., "Efficient Data Partitioning Model for Heterogeneous Graphs in the Cloud," in *ACM/IEEE SC*, 2013.

[71] Lee, K. and Liu, L., "Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning," *Proc. VLDB Endow.*, vol. 6, pp. 1894–1905, Sept. 2013.

[72] Leskovec, J., Kleinberg, J., and Faloutsos, C., "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *KDD*, 2005.

[73] Li, W., Serdyukov, P., de Vries, A. P., Eickhoff, C., and Larson, M., "The where in the tweet," in *CIKM*, 2011.

[74] Liao, H., Han, J., and Fang, J., "Multi-dimensional Index on Hadoop Distributed File System," in *NAS*, 2010.

[75] Lin, J., Snow, R., and Morgan, W., "Smoothing techniques for adaptive online language models: topic tracking in tweet streams," in *KDD*, 2011.

[76] Liu, X., Han, J., Zhong, Y., Han, C., and He, X., "Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS," in *CLUSTER*, 2009.

[77] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M., "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, pp. 716–727, Apr. 2012.

[78] Lu, W., Shen, Y., Chen, S., and Ooi, B. C., "Efficient Processing of K Nearest Neighbor Joins Using MapReduce," *Proc. VLDB Endow.*, vol. 5, June 2012.

[79] Mahmud, J., Nichols, J., and Drews, C., "Where Is This Tweet From? Inferring Home Locations of Twitter Users," in *ICWSM*, 2012.

[80] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G., "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010.

[81] Manning, C. D., Raghavan, P., and Schtze, H., *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[82] McCallum, A. K., "Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering." http://www.cs.cmu.edu/ mccallum/bow, 1996.

[83] Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P., and Bhattacharjee, B., "Measurement and Analysis of Online Social Networks," in *IMC*, 2007.

[84] Mouratidis, K., Yiu, M. L., Papadias, D., and Mamoulis, N., "Continuous nearest neighbor monitoring in road networks," in *VLDB*, 2006.

[85] Murugappan, A. and Liu, L., "An Energy Efficient Middleware Architecture for Processing Spatial Alarms on Mobile Clients," *Mob. Netw. Appl.*, vol. 15, pp. 543–561, August 2010.

[86] Neumann, T. and Weikum, G., "The RDF-3X engine for scalable management of RDF data," *VLDBJ*, 2010.

[87] Noulas, A., Scellato, S., Mascolo, C., and Pontil, M., "An Empirical Study of Geographic User Activity Patterns in Foursquare," in *ICWSM*, 2011.

[88] Owens, A., Seaborne, A., Gibbins, N., and others, "Clustered TDB: A Clustered Triple Store for Jena," 2008.

[89] Papadias, D., Sellis, T., Theodoridis, Y., and Egenhofer, M. J., "Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees," *SIGMOD Rec.*, vol. 24, May 1995.

[90] Papadias, D., Zhang, J., Mamoulis, N., and Tao, Y., "Query processing in spatial network databases," in *VLDB*, 2003.

[91] Pearce, R., Gokhale, M., and Amato, N. M., "Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory," in *SC*, 2010.

[92] Pennacchiotti, M. and Popescu, A.-M., "Democrats, republicans and starbucks afficionados: user classification in twitter," in *KDD*, 2011.

[93] Perry, M., Jain, P., and Sheth, A. P., "Sparql-st: Extending sparql to support spatiotemporal queries," in *Geospatial semantics and the semantic web*, Springer, 2011.

[94] Pesti, P., Liu, L., Bamba, B., Iyengar, A., and Weber, M., "RoadTrack: scaling location updates for mobile clients on road networks with query awareness," *Proc. VLDB Endow.*, vol. 3, 2010.

[95] Piorkowski, M., Sarafijanovoc-Djukic, N., and Grossglauser, M., "A Parsimonious Model of Mobile Partitioned Networks with Clustering," in *COMSNETS*, 2009.

[96] Prabhakar, S., Xia, Y., Kalashnikov, D. V., Aref, W. G., and Hambrusch, S. E., "Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects," *IEEE Trans. Comput.*, vol. 51, pp. 1124–1140, Oct. 2002.

[97] Rohloff, K. and Schantz, R. E., "Clause-iteration with MapReduce to scalably query datagraphs in the SHARD graph-store," in *DIDC*, 2011.

[98] Roy, A., Mihailovic, I., and Zwaenepoel, W., "X-Stream: Edge-centric Graph Processing Using Streaming Partitions," in *SOSP*, 2013.

[99] Sakaki, T., Okazaki, M., and Matsuo, Y., "Earthquake shakes Twitter users: real-time event detection by social sensors," in *WWW*, 2010.

[100] Salihoglu, S. and Widom, J., "GPS: A Graph Processing System," in *SSDBM*, 2013.

[101] Schmidt, M., Hornung, T., Lausen, G., and others, "SP^{2}Bench: A SPARQL Performance Benchmark," in *ICDE*, 2009.

[102] Sebastiani, F., "Machine learning in automated text categorization," *ACM Comput. Surv.*, vol. 34, no. 1, pp. 1–47, 2002.

[103] Seidl, T. and Kriegel, H.-P., "Optimal multi-step k-nearest neighbor search," in *SIGMOD*, 1998.

[104] Sellis, T. K., Roussopoulos, N., and Faloutsos, C., "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects," in *VLDB*, 1987.

[105] Serdyukov, P., Murdock, V., and van Zwol, R., "Placing flickr photos on a map," in *SIGIR*, 2009.

[106] Shao, B., Wang, H., and Li, Y., "Trinity: A Distributed Graph Engine on a Memory Cloud," in *SIGMOD*, 2013.

[107] Song, Z. and Roussopoulos, N., "K-Nearest Neighbor Search for Moving Query Point," in *SSTD*, 2001.

[108] Stanton, I. and Kliot, G., "Streaming Graph Partitioning for Large Distributed Graphs," in *KDD*, 2012.

[109] Starbird, K. and Palen, L., "(How) will the revolution be retweeted?: information diffusion and the 2011 Egyptian uprising," in *CSCW*, 2012.

[110] Suri, S. and Vassilvitskii, S., "Counting triangles and the curse of the last reducer," in *WWW*, 2011.

[111] Tao, Y., Papadias, D., and Shen, Q., "Continuous nearest neighbor search," in *VLDB*, 2002.

[112] Thiagarajan, A., Biagioni, J., Gerlich, T., and Eriksson, J., "Cooperative transit tracking using smart-phones," in *SenSys*, 2010.

[113] Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J., "From" think like a vertex" to" think like a graph," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, 2013.

[114] Valiant, L. G., "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103–111, Aug. 1990.

[115] White, B., Lepreau, J., Stoller, L., Ricci, R., and others, "An integrated experimental environment for distributed systems and networks," in *OSDI*, 2002.

[116] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A., "An Integrated Experimental Environment for Distributed Systems and Networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, 2002.

[117] Xing, S., Shahabi, C., and Pan, B., "Continuous monitoring of nearest neighbors on land surface," *Proc. VLDB Endow.*, 2009.

[118] Yu, X., Pu, K. Q., and Koudas, N., "Monitoring k-Nearest Neighbor Queries over Moving Objects," in *ICDE*, 2005.

[119] Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W., and Liu, L., "TripleBit: a Fast and Compact System for Large Scale RDF Data," *Proceedings of the VLDB Endowment*, vol. 6, no. 7, 2013.

[120] Yuan, P., Zhang, W., Xie, C., Jin, H., Liu, L., and Lee, K., "Fast Iterative Graph Computation: A Path Centric Approach," in *SC*, 2014.

[121] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I., "Spark: Cluster Computing with Working Sets," in *HotCloud*, 2010.

[122] Zhang, C., Li, F., and Jestes, J., "Efficient Parallel kNN Joins for Large Data in MapReduce," in *EDBT*, 2012.

[123] Zhang, S., Han, J., Liu, Z., Wang, K., and Feng, S., "Spatial Queries Evaluation with MapReduce," in *GCC*, 2009.

[124] Zheng, D., Mhembere, D., Burns, R., Vogelstein, J., Priebe, C. E., and Szalay, A. S., "FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs," in *FAST*, 2015.

[125] Zheng, Y., Zhang, L., Xie, X., and Ma, W.-Y., "Mining Interesting Locations and Travel Sequences from GPS Trajectories," in *WWW*, 2009.